The Journal of Logic and Algebraic Programming 80 (2011) 392–426

Contents lists available at ScienceDirect

The Journal of Logic and Algebraic Programming

journal homepage: www.elsevier.com/locate/jlap

Structural operational semantics through context-dependent behaviour

Robert J. Colvin [a,b,*], Ian J. Hayes [a]

[a] The University of Queensland, Australia
[b] The Queensland Brain Institute, Australia

ARTICLE INFO

Article history:
Received 25 September 2009
Revised 10 May 2011
Accepted 17 May 2011
Available online 30 May 2011

ABSTRACT

We present a new approach to providing a structural operational semantics for imperative programming languages with concurrency and procedures. The approach is novel because we expose the building block operations—variable assignment and condition checking—in the labels on the transitions; these form the *context-dependent behaviour* of a program. Using this style results in two main advantages over standard formalisms for imperative programming language semantics: firstly, our individual transition rules are more concise, and secondly, we are able to more abstractly and intuitively describe the semantics of procedures, including by-value and by-reference parameters. Standard techniques in the literature tend to result in complex and hard-to-read rules for even simple language constructs when procedures and parameters are dealt with. Our semantics for procedures utilises the context-dependent behaviour in the transition label to handle variable name scoping, and defines the semantics of recursion without requiring additional rules. In contrast with Plotkin's seminal structural operational semantics paper, we do not use *locations* to describe some of the more complex language constructs. Novel aspects of the abstract syntax include local states (in contrast to a single global store), which simplifies the reasoning about local variables, and a command for dynamically renaming variables (in contrast to mapping variables to locations), which simplifies the reasoning about the effect of procedures on by-reference parameters.

1. Introduction

In 1981, Plotkin [32,33] described the semantics of an imperative programming language using *Structural Operational Semantics* (SOS). This description was considerably simpler than previous formulations, which contained a significant amount of implementation detail that got in the way of a higher-level understanding of the basic constructs. SOS has become one of the preferred methods for describing language semantics, from sequential programming languages [16,23,35,44] to process calculi [2,22,38,39].

In standard, or Plotkin-style, SOS approaches, the value of variables appears as a meta-level construct in all transition rules, and the environment, which includes procedure definitions and constants, is collected starting from the outermost scope, with inner environments overriding the outer. Then the effect of a basic command such as an assignment can be precisely determined because its context is supplied.

In this paper we employ a different approach to SOS, where the transition arrows are labelled by the state-based behaviour associated with the step, i.e., the labels explicitly specify testing or updating the values of state variables. We refer to the

* Corresponding author at: The University of Queensland, Australia.
E-mail address: robert@itee.uq.edu.au (R.J. Colvin)

1567-8326/$ - see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jlap.2011.05.001

information in the label as the *context-dependent behaviour* of the command being executed, where the *context* consists of the values of variables and procedure definitions. The context-dependent behaviour of a basic command is nondeterministic, in that different behaviour will occur in different contexts. For instance, the evaluation of expression x has as many possible answers as there are possible values for variable x. However, given a particular context, in which $x = 3$ (say), there is only one valid evaluation, and this is the behaviour selected by the rules. The nondeterminism is resolved neatly by the style of transition rules found in an operational semantics.

The concept of context-dependent behaviour appearing in transition labels is not new, and is the typical approach to specifying the operational semantics of process calculi [22,38]. However, in that setting, the context-dependent behaviour of a process, which is often called the *observable* behaviour, is usually an abstract event. In an imperative language setting, the context-dependent "events" are updates and tests of the value of variables.

The paper is organised into two main parts. In the first part, Sections 2–5, we describe in a small-step style the semantics of an imperative programming language with concurrency and procedures (including recursive procedures), that admits functions with side-effects to appear in expressions. In the second part, Sections 6–9, we examine some consequences of, and extensions to, the semantics, including an atomic command, a big-step semantics, a notion of program equivalence, as well as static binding and arrays. In Section 10 we compare our approach to that of Plotkin, and discuss other related work.

As the definition of the static semantics (e.g., type correctness) of our language would be no different to that of Plotkin, we do not address static semantics in this paper. By extension, we implicitly assume all commands and expressions appearing in our rules are well-formed. We present our language as untyped, although, as above, we assume that the rules for a typed language are straightforward extensions of those we provide, following similar principles to existing type systems [33,35,44].

2. Expression evaluation

In this section we introduce some of our basic syntax, in particular expressions (Section 2.1), and describe how they may be incrementally evaluated in a *small-step* style using operational semantics through context-dependent behaviour (Section 2.2). In the first part of the paper we use a small-step style for both expression evaluation and, later, command execution. The choice was made in favour of small-step for command evaluation because it admits the potential for non-terminating executions, as may be required for reactive programs. However, the rules in a big-step style are often more compact, and hence for completeness we give a big-step semantics in Section 7.

2.1. Basic syntax

Assume a set of values, *Val*, and a set of identifiers, *Ident*. We assume the set *Val* contains the standard boolean and integer values, as well as more complex values as required. The set *Ident* is a set of identifiers, which are used for variable names, and later for procedure names.

A *State* is a partial mapping from variable identifiers to values, $Ident \pfun Val$. We use the notation $\{x \mapsto v\}$ to represent the state that maps identifier x to the value v. The set of variables declared by a state σ are those in its domain, written $\mathrm{dom}(\sigma)$, e.g., $\mathrm{dom}(\{x \mapsto v\}) = \{x\}$.

Note there are two differences with Plotkin-style semantics with respect to the handling of values of variables: firstly, our states are partial, and secondly, we do not introduce another mapping to handle references to the same variable using locations (the *environment*).

The abstract syntax of an expression, $e \in Expr$, is given below, where $v \in Val$, $x \in Ident$, and $\sigma \in State$.

$$e ::= v \mid x \mid (e_1 + e_2) \mid (\textbf{let}\ \sigma\ \textbf{in}\ e)$$

An expression is either a value, a variable, the addition of two expressions, or a **let** expression. In a **let** expression the state σ may provide values for some of the variables in e, and is (part of) the *context* in which e appears. Note that for presentation purposes the **let** expression, in which variables are associated with values, is less general than the usual definition, in which variables are associated with expressions. The semantics for the latter definition is straightforward to define. A more general expression language is given in Section 5.

2.2. Semantics

In this section we give the transition rules for expression evaluation in a small-step style, where a 'step' may be looking up the value of a variable or calculating the effect of some operation on values. A big-step semantics is given in Section 7.

The semantics of expression evaluation is given through a labelled transition relation.

$$\longrightarrow : Label \to \mathbb{P}(Expr \times Expr)$$

It is defined as the least relation that satisfies the operational rules. A transition $e_1 \xrightarrow{\ell} e_2$ states that expression e_1 (partially) evaluates to expression e_2, provided that the label ℓ is satisfied. The label ℓ is the context-dependent behaviour, that is, the transition occurs provided ℓ holds in context. For expression evaluation, there are two possible forms for a label

Assume $x \in Ident \quad v, v_i \in Val \quad e, e_i \in Expr \quad \ell \in Label \quad \sigma \in State$

Rule 1 (**Evaluate variable**).

$$x \xrightarrow{x=v} v$$

Rule 2 (**Evaluate addition**).

$$(a)\ \frac{e_1 \xrightarrow{\ell} e_1'}{(e_1 + e_2) \xrightarrow{\ell} (e_1' + e_2)} \qquad (b)\ \frac{e_2 \xrightarrow{\ell} e_2'}{(v + e_2) \xrightarrow{\ell} (v + e_2')} \qquad (c)\ (v_1 + v_2) \xrightarrow{\tau} v_1 + v_2$$

Rule 3 (**Evaluate let expression**).

$$(a)\ (\textbf{let}\ \sigma\ \textbf{in}\ v) \xrightarrow{\tau} v \qquad (b)\ \frac{e \xrightarrow{\tau} e'}{(\textbf{let}\ \sigma\ \textbf{in}\ e) \xrightarrow{\tau} (\textbf{let}\ \sigma\ \textbf{in}\ e')}$$

$$(c)\ \frac{e \xrightarrow{x=v} e' \quad x \in \mathrm{dom}(\sigma) \quad \sigma(x) = v}{(\textbf{let}\ \sigma\ \textbf{in}\ e) \xrightarrow{\tau} (\textbf{let}\ \sigma\ \textbf{in}\ e')} \qquad (d)\ \frac{e \xrightarrow{x=v} e' \quad x \notin \mathrm{dom}(\sigma)}{(\textbf{let}\ \sigma\ \textbf{in}\ e) \xrightarrow{x=v} (\textbf{let}\ \sigma\ \textbf{in}\ e')}$$

Fig. 1. Small-step expression evaluation rules.

$\ell \in Label$: an internal step, τ, and an equality $x = v$, where $x \in Ident$ and $v \in Val$.

$$\ell ::= \tau \mid x = v \tag{1}$$

A transition with label τ is allowed in any context. A transition with label $x = v$ is allowed only in contexts in which x currently has the value v. When describing the semantics of commands, the syntax of labels will be extended to also describe updates of the state and the returned values from function calls.

The rules for defining expression evaluation are given in Fig. 1. Note that there is no rule corresponding to an expression which is a value, v, because no evaluation is required. A variable x evaluates to a value v provided $x = v$ in context (Rule 1). This transition defines a relationship between every variable and every value. For instance, assume that there are only two variables, $Ident = \{x, y\}$, and two values, $Val = \{0, 1\}$. Then the complete set of possible transitions is as follows.

$$x \xrightarrow{x=0} 0 \qquad x \xrightarrow{x=1} 1 \qquad y \xrightarrow{y=0} 0 \qquad y \xrightarrow{y=1} 1 \tag{2}$$

The rules for evaluating a binary addition enforce a strict left-to-right evaluation order, where the left-most operand must be fully evaluated to a value before the right-most operand is evaluated [1] (Rule 2(a) and (b)). The context-dependent behaviour of e_1 when being evaluated, ℓ, is the context-dependent behaviour of the expression $(e_1 + e_2)$ when being evaluated, and similarly for e_2.

When both operands have been evaluated to values, the sum is calculated (Rule 2(c)). Note the distinction between the symbol '+' on the left of the transition arrow, which is a syntactic construct, and the symbol '+' on the right, which denotes the semantics of addition. This form of transition rule may be used to specify the evaluation of other expressions; in particular we assume similar rules exist for calculating subtraction, multiplication, exponentiation, and (in)equalities.

As an example, consider the evaluation of expression $x + y$ in the state space with two values and two variables, i.e., where the complete set of transitions allowed via Rule 1 is given by (2). From Rule 2(a) and Rule 1 we have two possible evaluations.

$$x + y \xrightarrow{x=0} 0 + y \qquad x + y \xrightarrow{x=1} 1 + y$$

Taking the case where $x = 0$ in context (the mechanics of this are explained below), the evaluation continues via Rule 2(b) and Rule 1.

$$0 + y \xrightarrow{y=0} 0 + 0 \qquad 0 + y \xrightarrow{y=1} 0 + 1$$

[1] Replacing both occurrences of v by e_1 in Rule 2(b) would allow any order of evaluation.

Taking the case where $y = 1$ in context, we have the following transition from Rule 2(c) that completes the evaluation.

$$0 + 1 \xrightarrow{\tau} 1$$

The τ label on the transition indicates that, as expected, the expression 0+1 evaluates to 1 in any context. No more evaluation is possible, since we have reduced the expression to the (basic) value 1. Collecting the above steps gives the following.

$$x + y \xrightarrow{x=0} 0 + y \xrightarrow{y=1} 0 + 1 \xrightarrow{\tau} 1$$

Now let us consider evaluating a **let** expression. Rule 3(a) applies when the expression within the **let** has already been fully evaluated to a value v, and therefore the state σ is no longer required and is eliminated. Rule 3(b) applies when the evaluation step has a label of τ. Such a step applies in any context, and the state σ is therefore irrelevant. For example, the following two transitions are justified by Rule 2(c) & Rule 3(b), and Rule 3(a), respectively, for any σ.

$$(\textbf{let } \sigma \textbf{ in } 0 + 1) \xrightarrow{\tau} (\textbf{let } \sigma \textbf{ in } 1) \xrightarrow{\tau} 1$$

Rule 3(c) and Rule 3(d) handle the more interesting case where the label is of the form $x = v$. We require two rules, depending on whether or not x is contained in the domain of σ, that is, whether x is local to the **let** expression.

Let us consider the evaluation of the expression x in a state where $x = 0$. The following transition is justified by Rule 1 and Rule 3(c).

$$(\textbf{let } \{x \mapsto 0\} \textbf{ in } x) \xrightarrow{\tau} (\textbf{let } \{x \mapsto 0\} \textbf{ in } 0)$$

Although, as shown in (2), there are two possible evaluations of x (to either 0 or 1), when a state σ is provided only the value $\sigma(x)$ is allowed by the premise "$\sigma(x) = v$" of Rule 3(c). This constraint eliminates the nondeterminism inherent in Rule 1. The transition $x \xrightarrow{x=1} 1$ cannot be selected once placed in context $\{x \mapsto 0\}$. Note that the transition label becomes τ, because there is no dependence on an outer context.

When x is not in the domain of σ, as given by Rule 3(d), the label $x = v$ is preserved as it is dependent on an outer context.

As a demonstration, consider the step-by-step evaluation of the expression

$$(\textbf{let } \{y \mapsto 1\} \textbf{ in } (\textbf{let } \{x \mapsto 0\} \textbf{ in } x + y)). \tag{3}$$

There are five transitions which are shown below, the first four of which are justified in Fig. 2. First x is evaluated, then y, then their sum is calculated, and finally the two now-redundant states are eliminated.

$$\begin{aligned}
&(\textbf{let } \{y \mapsto 1\} \textbf{ in } (\textbf{let } \{x \mapsto 0\} \textbf{ in } x + y)) \\
\xrightarrow{\tau}&\text{ See (4)} \\
&(\textbf{let } \{y \mapsto 1\} \textbf{ in } (\textbf{let } \{x \mapsto 0\} \textbf{ in } 0 + y)) \\
\xrightarrow{\tau}&\text{ See (5)} \\
&(\textbf{let } \{y \mapsto 1\} \textbf{ in } (\textbf{let } \{x \mapsto 0\} \textbf{ in } 0 + 1)) \\
\xrightarrow{\tau}&\text{ See (6)} \\
&(\textbf{let } \{y \mapsto 1\} \textbf{ in } (\textbf{let } \{x \mapsto 0\} \textbf{ in } 1)) \\
\xrightarrow{\tau}&\text{ See (7)} \\
&(\textbf{let } \{y \mapsto 1\} \textbf{ in } 1) \\
\xrightarrow{\tau}&\text{ Rule 3(a)} \\
&1
\end{aligned}$$

Notice that in Fig. 2 each transition can be read intuitively by treating the label as the guard for the transition to occur. Other choices can be made when applying Rule 1 at the top of the inferences in (4) and (5), but such choices would not result in a successful application of Rule 3(c), and hence would not form part of a valid inference.

Below we demonstrate how the semantics handles variable name reuse in separate (non-overlapping) parts of an expression. First consider evaluating the addition of two let expressions that declare the same variable: (**let** $\{x \mapsto 0\}$ **in** x) + (**let** $\{x \mapsto 1\}$ **in** x).

$$\dfrac{\dfrac{x \xrightarrow{x=0} 0}{(\textbf{let } \{x \mapsto 0\} \textbf{ in } x) \xrightarrow{\tau} (\textbf{let } \{x \mapsto 0\} \textbf{ in } 0)}\ \textit{Rule 3(c)}}{\begin{array}{l}(\textbf{let } \{x \mapsto 0\} \textbf{ in } x) + \\ (\textbf{let } \{x \mapsto 1\} \textbf{ in } x)\end{array} \xrightarrow{\tau} \begin{array}{l}(\textbf{let } \{x \mapsto 0\} \textbf{ in } 0) + \\ (\textbf{let } \{x \mapsto 1\} \textbf{ in } x)\end{array}}\ \textit{Rule 2(a)} \tag{8}$$

$$\dfrac{\dfrac{\dfrac{}{x \xrightarrow{x=0} 0}\ \textit{Rule 1}}{x + y \xrightarrow{x=0} 0 + y}\ \textit{Rule 2(a)}}{(\textbf{let}\ \{x \mapsto 0\}\ \textbf{in}\ x + y) \xrightarrow{\tau} (\textbf{let}\ \{x \mapsto 0\}\ \textbf{in}\ 0 + y)}\ \textit{Rule 3(c)}}{\begin{array}{l}(\textbf{let}\ \{y \mapsto 1\}\ \textbf{in} \\ \quad (\textbf{let}\ \{x \mapsto 0\}\ \textbf{in}\ x + y))\end{array} \xrightarrow{\tau} \begin{array}{l}(\textbf{let}\ \{y \mapsto 1\}\ \textbf{in} \\ \quad (\textbf{let}\ \{x \mapsto 0\}\ \textbf{in}\ 0 + y))\end{array}}\ \textit{Rule 3(b)} \tag{4}$$

$$\dfrac{\dfrac{\dfrac{}{y \xrightarrow{y=1} 1}\ \textit{Rule 1}}{0 + y \xrightarrow{y=1} 0 + 1}\ \textit{Rule 2(b)}}{(\textbf{let}\ \{x \mapsto 0\}\ \textbf{in}\ 0 + y) \xrightarrow{y=1} (\textbf{let}\ \{x \mapsto 0\}\ \textbf{in}\ 0 + 1)}\ \textit{Rule 3(d)}}{\begin{array}{l}(\textbf{let}\ \{y \mapsto 1\}\ \textbf{in} \\ \quad (\textbf{let}\ \{x \mapsto 0\}\ \textbf{in}\ 0 + y))\end{array} \xrightarrow{\tau} \begin{array}{l}(\textbf{let}\ \{y \mapsto 1\}\ \textbf{in} \\ \quad (\textbf{let}\ \{x \mapsto 0\}\ \textbf{in}\ 0 + 1))\end{array}}\ \textit{Rule 3(c)} \tag{5}$$

$$\dfrac{\dfrac{\dfrac{}{0 + 1 \xrightarrow{\tau} 1}\ \textit{Rule 2(c)}}{(\textbf{let}\ \{x \mapsto 0\}\ \textbf{in}\ 0 + 1) \xrightarrow{\tau} (\textbf{let}\ \{x \mapsto 0\}\ \textbf{in}\ 1)}\ \textit{Rule 3(b)}}{\begin{array}{l}(\textbf{let}\ \{y \mapsto 1\}\ \textbf{in} \\ \quad (\textbf{let}\ \{x \mapsto 0\}\ \textbf{in}\ 0 + 1))\end{array} \xrightarrow{\tau} \begin{array}{l}(\textbf{let}\ \{y \mapsto 1\}\ \textbf{in} \\ \quad (\textbf{let}\ \{x \mapsto 0\}\ \textbf{in}\ 1))\end{array}}\ \textit{Rule 3(b)} \tag{6}$$

$$\dfrac{\dfrac{}{(\textbf{let}\ \{x \mapsto 0\}\ \textbf{in}\ 1) \xrightarrow{\tau} 1}\ \textit{Rule 3(a)}}{(\textbf{let}\ \{y \mapsto 1\}\ \textbf{in}\ (\textbf{let}\ \{x \mapsto 0\}\ \textbf{in}\ 1)) \xrightarrow{\tau} (\textbf{let}\ \{y \mapsto 1\}\ \textbf{in}\ 1)}\ \textit{Rule 3(b)} \tag{7}$$

Fig. 2. Expression evaluation steps.

The test on the value of the left-most reference to x does not appear in the transition label at the outer level—it is resolved locally to the left-most expression, and does not cause interference with the second part of the expression which declares its own local variable x. Hence we have the following complete derivation for the expression.

$(\textbf{let}\ \{x \mapsto 0\}\ \textbf{in}\ x) + (\textbf{let}\ \{x \mapsto 1\}\ \textbf{in}\ x)$
$\xrightarrow{\tau}$ See (8)
$(\textbf{let}\ \{x \mapsto 0\}\ \textbf{in}\ 0) + (\textbf{let}\ \{x \mapsto 1\}\ \textbf{in}\ x)$
$\xrightarrow{\tau}$ Rule 3(a), Rule 2(a)
$0 + (\textbf{let}\ \{x \mapsto 1\}\ \textbf{in}\ x)$
$\xrightarrow{\tau}$ Similar reasoning to (8)
$0 + (\textbf{let}\ \{x \mapsto 1\}\ \textbf{in}\ 1)$
$\xrightarrow{\tau}$ Rule 3(a), Rule 2(a)
$0 + 1$
$\xrightarrow{\tau}$ Rule 2(c)
1

Similarly, if the same identifier appears in nested scopes, the innermost state takes precedence, as shown in the following evaluation.

$(\textbf{let}\ \{x \mapsto 0\}\ \textbf{in}\ (\textbf{let}\ \{x \mapsto 1\}\ \textbf{in}\ x))$
$\xrightarrow{\tau}$ Rule 1, Rule 3(c), Rule 3(b)

$$
\begin{array}{l}
\quad (\textbf{let}\ \{x \mapsto 0\}\ \textbf{in}\ (\textbf{let}\ \{x \mapsto 1\}\ \textbf{in}\ 1)) \\
\xrightarrow{\tau} \text{Rule 3(a), Rule 3(b)} \\
\quad (\textbf{let}\ \{x \mapsto 0\}\ \textbf{in}\ 1) \\
\xrightarrow{\tau} \text{Rule 3(a)} \\
\quad 1
\end{array}
$$

3. Command execution

In this section we describe the semantics of an imperative programming language with concurrency.

3.1. Syntax

The abstract syntax of a command $c \in Cmd$ is described below, where $x \in Ident$ and $e, b \in Expr$ (we use e for expressions of any type and b for expressions of boolean type).

$$c ::= \textbf{nil} \mid (x := e) \mid (c_1 \ ; \ c_2) \mid (c_1 \parallel c_2) \mid (\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2) \mid (\textbf{while}\ b\ \textbf{do}\ c) \mid (\textbf{state}\ \sigma \bullet c)$$

The special command **nil** indicates a terminated command. It can partake in no further action. An assignment $x := e$ is the standard assignment command. For the moment we assume $x \in Ident$ and e is an expression as defined in the previous section, although in Section 5 we give a richer expression syntax for e, and in Section 9.2 consider other possibilities for the left-hand side. The command "$c_1 \ ; \ c_2$" denotes sequential composition, and we assume an interleaving semantics for a parallel composition $c_1 \parallel c_2$. The conditional and while commands are standard.

The *local state* command $(\textbf{state}\ \sigma \bullet c)$ declares variables in the domain of σ to be 'local' to c. This abstract command type may be introduced through elaborating a variable declaration in the concrete syntax, e.g., if T is a type, a value v from T is chosen nondeterministically to initialise x.

$$\frac{v \in T}{(\mathtt{var}\ x{:}\,T \ ; \ c) \xrightarrow{\tau} (\textbf{state}\ \{x \mapsto v\} \bullet c)}$$

In the Plotkin-style approach, x is mapped locally to some location, which is globally mapped to the initial value.

Constant declarations are treated in our abstract syntax as members of a local state. As such, their value may be tested in the same manner as mutable variables; it is simply the case that they are never updated. In most concrete syntax, however, constants and variables are distinct.

3.2. Transitions

The semantics are defined with respect to a transition relation.

$$\longrightarrow : Label \to \mathbb{P}(Cmd \times Cmd)$$

It is the least relation that satisfies the operational rules. The syntax of labels (1) is extended to include assignments of a value v to a variable x.

$$\ell ::= \tau \mid x = v \mid x := v$$

The new label type, $x := v$, requires that the context updates x to the value v. Hence, a transition $c \xrightarrow{x := v} c'$ states that command c transitions to c' and has the behaviour of setting x to v in some outer context. Note that we do not syntactically distinguish transitions on expressions from transitions on commands, but we ensure the correct relation is always clear from the intended types. As an abbreviation, in the sequel we omit labels of τ from the transition arrow.

3.3. Standard commands

The semantics of the language appears in Fig. 3. The command **nil** can take no transitions, and therefore there is no corresponding rule. An assignment (Rule 4) first evaluates the expression e (eventually) to a value, then exposes the assignment of that value in the label. Note that for simplicity we assume that, although the evaluation of e is non-atomic, the final update of x is atomic, regardless of the complexity of the value v that is evaluated. The sequential composition rule (Rule 5) is standard: if the first command can take some step then the composition also can, and when the first command has terminated the second may execute. Rule 6 states that concurrent processes interleave their operations; a terminated concurrent process may be eliminated (Rule 6(c) and (d)). Rule 7 states that a conditional first evaluates its guard (eventually) to a boolean value, and then chooses either c_1 or c_2 depending on the result of the evaluation. An iteration command is unfolded into a conditional command (Rule 8); if the condition b evaluates to *false* the iteration terminates, otherwise, if b evaluates to

Assume $x \in Ident \quad v \in Val \quad e, b \in Expr \quad c, c_i \in Cmd \quad \ell \in Label$

Rule 4 (**Assignment**).

$$(a)\ \frac{e \xrightarrow{\ell} e'}{(x := e) \xrightarrow{\ell} (x := e')} \qquad (b)\ (x := v) \xrightarrow{x := v} \mathbf{nil}$$

Rule 5 (**Sequential composition**).

$$(a)\ \frac{c_1 \xrightarrow{\ell} c_1'}{(c_1 \,;\, c_2) \xrightarrow{\ell} (c_1' \,;\, c_2)} \qquad (b)\ (\mathbf{nil} \,;\, c_2) \longrightarrow c_2$$

Rule 6 (**Interleave**).

$$(a)\ \frac{c_1 \xrightarrow{\ell} c_1'}{(c_1 \parallel c_2) \xrightarrow{\ell} (c_1' \parallel c_2)} \qquad (b)\ \frac{c_2 \xrightarrow{\ell} c_2'}{(c_1 \parallel c_2) \xrightarrow{\ell} (c_1 \parallel c_2')} \qquad (c)\ (\mathbf{nil} \parallel c) \longrightarrow c \qquad (d)\ (c \parallel \mathbf{nil}) \longrightarrow c$$

Rule 7 (**Conditional**).

$$(a)\ \frac{b \xrightarrow{\ell} b'}{(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) \xrightarrow{\ell} (\mathbf{if}\ b'\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2)}$$

$$(b)\ (\mathbf{if}\ \mathit{true}\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) \longrightarrow c_1 \qquad (c)\ (\mathbf{if}\ \mathit{false}\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) \longrightarrow c_2$$

Rule 8 (**While**).

$$(\mathbf{while}\ b\ \mathbf{do}\ c) \longrightarrow (\mathbf{if}\ b\ \mathbf{then}\ (c \,;\, (\mathbf{while}\ b\ \mathbf{do}\ c))\ \mathbf{else}\ \mathbf{nil})$$

Fig. 3. Command execution rules.

true, the body of the iteration c is executed followed by the execution of the complete iteration (which will require a further unfolding of the iteration using Rule 8, and so on).

3.4. State-based rules

The rules for a local state are given in Fig. 4. Rule 9 states that a local state is eliminated when its command terminates; this is similar to Rule 3(a). Rule 10 states that if c can transition independently of the context, then so can (**state** $\sigma \bullet c$). Being context-independent, there is no constraint on σ (cf. Rule 3(b)). Rule 11(a) applies when the command c requires $x = v$ to be true to transition to c', and x is in the domain of σ. Syntax aside, this rule is effectively the same as Rule 3(c)—a transition is possible only when v is the value for x in σ. The behaviour no longer depends on the context. Rule 11(b) applies when x is not in the domain of σ. Syntax aside, this rule is effectively the same as Rule 3(d). Since x is not in the domain of the state, the context-dependent behaviour remains the same.

The rules for assignments are the more interesting cases. Rule 12(a) applies when the command c modifies the local state according to the label $x := v$. If x is local to σ, then the value of x in σ is updated to v (we use Plotkin's notation for an updated state, $\sigma[x \mapsto v]$). Because the update is to a local variable, the transition is not context-dependent and hence becomes internal (that is, the label becomes τ, which is by convention omitted from the transition arrow). For example, an assignment of a value to a local variable results in a change in the local state.

$$(\mathbf{state}\ \{x \mapsto 1\} \bullet x := 0) \longrightarrow (\mathbf{state}\ \{x \mapsto 0\} \bullet \mathbf{nil})$$

Rule 12(b) applies when c updates a variable outside the local state. In this case there is no change in σ, and the label remains the same, e.g.,

$$(\mathbf{state}\ \{x \mapsto 1\} \bullet y := 0) \xrightarrow{y := 0} (\mathbf{state}\ \{x \mapsto 1\} \bullet \mathbf{nil}).$$

Rule 9 (**Eliminate state**).

$$(\textbf{state}\ \sigma\ \bullet\ \textbf{nil}) \longrightarrow \textbf{nil}$$

Rule 10 (**State – context independent**).

$$\frac{c \longrightarrow c'}{(\textbf{state}\ \sigma\ \bullet\ c) \longrightarrow (\textbf{state}\ \sigma\ \bullet\ c')}$$

Rule 11 (**State – guard**).

$$(a)\ \frac{c \xrightarrow{x=v} c' \quad x \in \mathrm{dom}(\sigma) \quad \sigma(x) = v}{(\textbf{state}\ \sigma\ \bullet\ c) \longrightarrow (\textbf{state}\ \sigma\ \bullet\ c')} \qquad (b)\ \frac{c \xrightarrow{x=v} c' \quad x \notin \mathrm{dom}(\sigma)}{(\textbf{state}\ \sigma\ \bullet\ c) \xrightarrow{x=v} (\textbf{state}\ \sigma\ \bullet\ c')}$$

Rule 12 (**State – assignment**).

$$(a)\ \frac{c \xrightarrow{x:=v} c' \quad x \in \mathrm{dom}(\sigma)}{(\textbf{state}\ \sigma\ \bullet\ c) \longrightarrow (\textbf{state}\ \sigma[x \mapsto v]\ \bullet\ c')} \qquad (b)\ \frac{c \xrightarrow{x:=v} c' \quad x \notin \mathrm{dom}(\sigma)}{(\textbf{state}\ \sigma\ \bullet\ c) \xrightarrow{x:=v} (\textbf{state}\ \sigma\ \bullet\ c')}$$

Fig. 4. Local state execution.

Rule 13 (**Let – generalised**).

$$\frac{e \xrightarrow{\ell} e' \quad \mathit{consistent}(\ell, \sigma)}{(\textbf{let}\ \sigma\ \textbf{in}\ e) \xrightarrow{\ell[\sigma]} (\textbf{let}\ \sigma[\ell]\ \textbf{in}\ e')}$$

Rule 14 (**State – generalised**).

$$\frac{c \xrightarrow{\ell} c' \quad \mathit{consistent}(\ell, \sigma)}{(\textbf{state}\ \sigma\ \bullet\ c) \xrightarrow{\ell[\sigma]} (\textbf{state}\ \sigma[\ell]\ \bullet\ c')}$$

Definition 9 (**Label and state effects**). For $\ell \in \mathit{Label}$ and $\sigma \in \mathit{State}$,

$$\ell[\sigma] = \begin{cases} \tau & \text{if } \ell \text{ is } x = v \text{ and } x \in \mathrm{dom}(\sigma) \\ \tau & \text{if } \ell \text{ is } x := v \text{ and } x \in \mathrm{dom}(\sigma) \\ \ell & \text{otherwise} \end{cases}$$

$$\sigma[\ell] = \begin{cases} \sigma[x \mapsto v] & \text{if } \ell \text{ is } x := v \text{ and } x \in \mathrm{dom}(\sigma) \\ \sigma & \text{otherwise} \end{cases}$$

$$\mathit{consistent}(\ell, \sigma) \Leftrightarrow \begin{cases} \sigma(x) = v & \text{if } \ell \text{ is } x = v \text{ and } x \in \mathrm{dom}(\sigma) \\ \mathit{true} & \text{otherwise} \end{cases}$$

Fig. 5. Generalised **let** expression and local state rules.

To help show more clearly the principles involved in designing the rules for local states and **let** expressions, we present generalisations of Rule 3(b)–(d) and Rules 10–12. We note that each rule contains a possible change in the label, a possible change in the local state, and conditions on the relationship between the label and the state (in the premise).

The generalised rules and the relevant definitions are given in Fig. 5. In Rules 13 and 14 the label on the transition of the subexpression/subcommand, ℓ, is changed according to the local state, σ, which we write $\ell[\sigma]$, while the local state is modified according to ℓ, which we write $\sigma[\ell]$. The notation $\ell[\sigma]$ is defined so that a transition becomes silent (internal) if it refers to a variable in the local state, and is otherwise unchanged. The notation $\sigma[\ell]$ is defined so that the state is updated if a local variable is updated in the label, and is otherwise unchanged. The predicate $\mathit{consistent}(\ell, \sigma)$ is defined so that a guard

label $x = v$ where x is local is consistent with the local state only if $\sigma(x) = v$, while all other labels are consistent with any state.

Instantiating ℓ with each of the three possible label types, τ, $x = v$ and $x := v$, and further splitting the last two types into the cases where $x \in \mathrm{dom}(\sigma)$ and $x \notin \mathrm{dom}(\sigma)$, gives the relevant rules for let expressions and local state (Rule 3(a) and Rule 9 do not generalise in this manner). For instance, taking the case where ℓ is of the form $x := v$ and $x \in \mathrm{dom}(\sigma)$, we get $\ell[\sigma] = \tau, \sigma[\ell] = \sigma[x \mapsto v]$, and $\mathit{consistent}(\ell, \sigma) \Leftrightarrow \mathit{true}$. Making these substitutions into Rule 14 and including the case $x \in \mathrm{dom}(\sigma)$ as a premise, gives Rule 12(a) (since the *consistent* requirement is equivalent to *true* it is omitted). Similar reductions occur for the other label types and **let** expressions.

3.5. Lists and traces

Given some type T, the type **list** T is the set of all finite lists with element type T. If $\vec{a}_1$ and $\vec{a}_2$ are lists of some type, then '$\vec{a}_1.\vec{a}_2$' represents the concatenation of the two lists. Typically, a list of elements a, b, c is written $\langle a, b, c\rangle$, however, to reduce the notational overhead, where the types are obvious, we do not distinguish between an element and the singleton list containing exactly that element. That is, a may stand for either an element of type T or a singleton list of type **list** T, and hence, $a.\vec{a}$ constructs a non-empty list with first element a and tail the list $\vec{a}$, and $\vec{a}.a$ constructs a list with final element a. We use ϵ for the empty list.

The execution of a program by the small-step rules naturally gives rise to a list of consecutive labels; we call such a list a *trace*. The following definition formalises the construction of a trace by the repeated application of the transition rules for executing a command or evaluating an expression.

Definition 10 (Compound traces)**.** The relation $\Rightarrow$: $\mathit{Trace} \to \mathbb{P}(\mathit{Cmd} \times \mathit{Cmd})$ defines a series of steps in the relation $\longrightarrow$. We define the relation inductively.

$$c \overset{\epsilon}{\Longrightarrow} c \qquad c \overset{\ell.\ell s}{\Longrightarrow} c'' \Leftrightarrow (\exists\, c' \bullet c \overset{\ell}{\longrightarrow} c' \wedge c' \overset{\ell s}{\Longrightarrow} c'')$$

For brevity, we tend to omit internal labels τ from traces that annotate the '$\Rightarrow$' relation. Let τ^i, for $i \geq 0$, represents a trace of length i containing only τ labels (hence, $\tau^0 = \epsilon$). Then when we write $c \overset{\ell}{\Longrightarrow} c'$, we understand that there exists some $i, j \geq 0$ such that $c \overset{\tau^i}{\Longrightarrow} c_1 \overset{\ell}{\longrightarrow} c_2 \overset{\tau^j}{\Longrightarrow} c'$, that is, we leave internal steps implicit. By extension, we also let $c \overset{\ell s}{\Longrightarrow} c'$ abbreviate a series of transitions where any finite number of τ steps interleave with the labels in the trace ℓs. By convention, if all elements of a trace ℓs are τ, we omit it from the transition arrow.

The definition helps to compactly define the derived rules in Fig. 6, which are used to simplify later examples. For instance, Rule 15(a) is justified by the following execution, using Rule 4(b) & Rule 5(a) in the first step and Rule 5(b) in the second.

$$(x := v \;;\; c) \overset{x:=v}{\longrightarrow} (\mathbf{nil} \;;\; c) \longrightarrow c$$

Rule 15(b) follows similarly using Rule 12(a). Rule 16 summarises the evaluation of the condition in an **if** statement; by Rule 7(a) the condition is successively evaluated generating a trace, ℓs, and when *true* or *false* is the result then the command reduces to c_1 or c_2, respectively. Rule 17 follows from Rule 8 (unfolding the loop into a conditional) and Rule 16.

3.6. Examples

In this section we demonstrate the semantics on some examples.

3.6.1. Sequential computation

Consider a simple program that swaps the value of two variables, x and y, before executing some command c. The swap makes use of a local variable t, initially 0, which is not available inside c (we include c because otherwise the whole command would be equivalent to **nil**).

$$(\mathbf{state}\ \{x \mapsto 1, y \mapsto 2\} \bullet (\mathbf{state}\ \{t \mapsto 0\} \bullet t := x \;;\; x := y \;;\; y := t) \;;\; c) \tag{11}$$

The expression, x, of the first assignment, $t := x$, is evaluated in context to 1. The evaluation is shown in detail in Fig. 7. The evaluation requires straightforward propagation of the label $x = 1$ outwards through the program until it finds the relevant state which contains the value for x. We omit the details of such trivial expression evaluations throughout the rest of the paper.

The assignment $t := 1$ then transitions to **nil**, with the context-dependent behaviour being exactly $t := 1$. This is shown in detail in Fig. 7. As above, we omit the details of such trivial transitions.

The **nil**, which has taken the place of the assignment to t, is eliminated through Rule 5(b), and then the expression y in $x := y$ is evaluated in context to 2.

$$(\mathbf{state}\ \{x \mapsto 1, y \mapsto 2\} \bullet (\mathbf{state}\ \{t \mapsto 1\} \bullet x := 2 \;;\; y := t) \;;\; c)$$

Assume $\ell s \in Trace$

Rule 15 (**Sequential assignment**).

(a) $(x := v \ ; \ c) \xrightarrow{x := v} c$

(b) $(\textbf{state}\ \{x \mapsto v_1\} \bullet x := v_2 \ ; \ c) \Longrightarrow (\textbf{state}\ \{x \mapsto v_2\} \bullet c)$

Rule 16 (**Evaluate condition**).

(a) $$\frac{b \overset{\ell s}{\Longrightarrow} true}{(\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2) \overset{\ell s}{\Longrightarrow} c_1}$$ (b) $$\frac{b \overset{\ell s}{\Longrightarrow} false}{(\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2) \overset{\ell s}{\Longrightarrow} c_2}$$

Rule 17 (**Iterate while**).

(a) $$\frac{b \overset{\ell s}{\Longrightarrow} true}{(\textbf{while}\ b\ \textbf{do}\ c) \overset{\ell s}{\Longrightarrow} (c \ ; \ \textbf{while}\ b\ \textbf{do}\ c)}$$ (b) $$\frac{b \overset{\ell s}{\Longrightarrow} false}{(\textbf{while}\ b\ \textbf{do}\ c) \overset{\ell s}{\Longrightarrow} \textbf{nil}}$$

Fig. 6. Derived rules.

For space reasons we let d abbreviate the command $(x := y \ ; \ y := t)$, we abbreviate **state** to **st**, and we use a small font for the states themselves.

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{x \xrightarrow{x=1} 1}{t := x \xrightarrow{x=1} t := 1}}{t := x \ ; \ d \xrightarrow{x=1} t := 1 \ ; \ d}}{(\textbf{st}\ \{t \mapsto 0\} \bullet t := x \ ; \ d) \xrightarrow{x=1} (\textbf{st}\ \{t \mapsto 0\} \bullet t := 1 \ ; \ d)}}{(\textbf{st}\ \{t \mapsto 0\} \bullet t := x \ ; \ d) \ ; \ c \xrightarrow{x=1} (\textbf{st}\ \{t \mapsto 0\} \bullet t := 1 \ ; \ d) \ ; \ c}}{(\textbf{st}\ \{^{x \mapsto 1}_{y \mapsto 2}\} \bullet (\textbf{st}\ \{t \mapsto 0\} \bullet t := x \ ; \ d) \ ; \ c) \longrightarrow (\textbf{st}\ \{^{x \mapsto 1}_{y \mapsto 2}\} \bullet (\textbf{st}\ \{t \mapsto 0\} \bullet t := 1 \ ; \ d) \ ; \ c)}$$

The transition is justified, top-to-bottom, by Rule 1; Rule 4(a); Rule 5(a); Rule 11(b) (since $x \notin \mathrm{dom}(\{t \mapsto 0\})$); Rule 5(a); Rule 11(a).

$$\frac{\dfrac{\dfrac{\dfrac{t := 1 \xrightarrow{t := 1} \textbf{nil}}{t := 1 \ ; \ d \xrightarrow{t := 1} \textbf{nil} \ ; \ d}}{(\textbf{st}\ \{t \mapsto 0\} \bullet t := 1 \ ; \ d) \longrightarrow (\textbf{st}\ \{t \mapsto 1\} \bullet \textbf{nil} \ ; \ d)}}{(\textbf{st}\ \{t \mapsto 0\} \bullet t := 1 \ ; \ d) \ ; \ c \longrightarrow (\textbf{st}\ \{t \mapsto 1\} \bullet \textbf{nil} \ ; \ d) \ ; \ c}}{(\textbf{st}\ \{^{x \mapsto 1}_{y \mapsto 2}\} \bullet (\textbf{st}\ \{t \mapsto 0\} \bullet t := 1 \ ; \ d) \ ; \ c) \longrightarrow (\textbf{st}\ \{^{x \mapsto 1}_{y \mapsto 2}\} \bullet (\textbf{st}\ \{t \mapsto 1\} \bullet \textbf{nil} \ ; \ d) \ ; \ c)}$$

The transition is justified, top-to-bottom, by Rule 4(b); Rule 5(a); Rule 12(a); Rule 5(a); Rule 10.

Fig. 7. Transitions for (11).

As above, the assignment transitions to **nil**, with the context-dependent behaviour $x := 2$. This is passed through the inner state via Rule 12(b), and then results in a change in the outer state through Rule 12(a).

$(\textbf{state}\ \{x \mapsto 2, y \mapsto 2\} \bullet (\textbf{state}\ \{t \mapsto 1\} \bullet \textbf{nil} \ ; \ y := t) \ ; \ c)$

Let WH abbreviate (**while** $a > 0$ **do** $b := a + b \ ; \ a := a - 1$).

$$
\begin{array}{ll}
 & (\textbf{state}\ \{a \mapsto 2\} \bullet b := 0 \ ; \ WH) \\
\xrightarrow{b:=0} & \text{Rule 15(a), Rule 12(b)} \\
 & (\textbf{state}\ \{a \mapsto 2\} \bullet WH) \\
\Longrightarrow & \text{Expression evaluation, Rule 17(a), Rule 11(a)} \\
 & (\textbf{state}\ \{a \mapsto 2\} \bullet b := a + b \ ; \ a := a - 1 \ ; \ WH) \\
\xrightarrow{b=0} & \text{Expression evaluation} \\
 & (\textbf{state}\ \{a \mapsto 2\} \bullet b := 2 \ ; \ a := a - 1 \ ; \ WH) \\
\xrightarrow{b:=2} & \text{Rule 15(a), Rule 12(b)} \\
 & (\textbf{state}\ \{a \mapsto 2\} \bullet a := a - 1 \ ; \ WH) \\
\Longrightarrow & \text{Expression evaluation, Rule 15(b)} \\
 & (\textbf{state}\ \{a \mapsto 1\} \bullet WH) \\
\Longrightarrow & \text{Expression evaluation, Rule 17(a), Rule 11(a)} \\
 & (\textbf{state}\ \{a \mapsto 1\} \bullet b := a + b \ ; \ a := a - 1 \ ; \ WH) \\
\xrightarrow{b=2} & \text{Expression evaluation} \\
 & (\textbf{state}\ \{a \mapsto 1\} \bullet b := 3 \ ; \ a := a - 1 \ ; \ WH) \\
\xrightarrow{b:=3} & \text{Rule 15(a), Rule 12(b)} \\
 & (\textbf{state}\ \{a \mapsto 1\} \bullet a := a - 1 \ ; \ WH) \\
\Longrightarrow & \text{Expression evaluation, Rule 15(b)} \\
 & (\textbf{state}\ \{a \mapsto 0\} \bullet WH) \\
\Longrightarrow & \text{Expression evaluation, Rule 17(b), Rule 11} \\
 & (\textbf{state}\ \{a \mapsto 0\} \bullet \textbf{nil}) \\
\longrightarrow & \text{Rule 9} \\
 & \textbf{nil}
\end{array}
$$

Fig. 8. An execution of (12).

After eliminating the **nil** and evaluating the expression t to 1, the remaining assignment results in an update to y in the outer state.

$$(\textbf{state}\ \{x \mapsto 2, y \mapsto 1\} \bullet (\textbf{state}\ \{t \mapsto 1\} \bullet \textbf{nil}) \ ; \ c)$$

The inner state is no longer necessary, and is eliminated through Rules 9 and 5(b), resulting in the following command where the values of x and y have been swapped and c is ready to execute.

$$(\textbf{state}\ \{x \mapsto 2, y \mapsto 1\} \bullet c)$$

3.6.2. Iteration

Consider the execution of a iteration that calculates $\sum_{i=1}^{N} i$. The value N is the initial value of local variable a, and the command stores both intermediate results and the final value in some non-local variable b. In the case shown, the initial value of a is 2, i.e., $N = 2$, and hence the final value of b is 3.

$$
\begin{array}{l}
(\textbf{state}\ \{a \mapsto 2\} \bullet \\
\quad b := 0 \ ; \\
\quad (\textbf{while}\ a > 0\ \textbf{do} \\
\qquad b := a + b \ ; \\
\qquad a := a - 1))
\end{array}
\tag{12}
$$

The execution of (12) is given in Fig. 8. The steps are collected in their entirety below.

$$(12) \xrightarrow{(b:=0).(b=0).(b:=2).(b=2).(b:=3)} \textbf{nil}$$

The above trace leaves the internal steps (τ labels) implicit; these are accesses of local variable a, or loop unfolding steps.

Note that the above trace cannot be simplified, because it is context-dependent. For example, it is possible (although not desirable) to put this program in parallel with another program that also updates b, and hence may interleave steps that lead to other traces (because b may be evaluated differently).

3.6.3. Concurrency

Now we give the execution of a concurrent program. The commands share a non-local variable y and each has its own local variable x.

$$(\textbf{state}\ \{x \mapsto 0\} \bullet y := x) \quad \| \quad (\textbf{state}\ \{x \mapsto 1\} \bullet y := x) \tag{13}$$

Assuming that there are no further concurrent processes also modifying y, the final value of y will be either 0 or 1, depending on the order of execution.

Taking the first branch, from Rule 12 we have

$$\begin{aligned} & (\textbf{state}\ \{x \mapsto 0\} \bullet y := x) \\ \longrightarrow\ & (\textbf{state}\ \{x \mapsto 0\} \bullet y := 0) \\ \xrightarrow{y := 0}\ & (\textbf{state}\ \{x \mapsto 0\} \bullet \textbf{nil}) \\ \longrightarrow\ & \textbf{nil} \end{aligned}$$

Note that x does not appear in the labels, as it is local. A similar trace holds for the second branch. Outside of the parallel composition the name spaces are distinct, and therefore the reused variable name does not affect the computation.

Rule 6(a) and (b) introduces nondeterminism in which of the branches will execute, since their steps may be interleaved. Hence, collecting the possible traces of context-dependent behaviour of the command (13), we find two possibilities:

$$(y := 0).(y := 1) \qquad \text{and} \qquad (y := 1).(y := 0).$$

We have included the parallel operator in our language, partly because concurrency is increasingly prevalent in modern programming languages, and partly to demonstrate that our encapsulated state commands handle multiple instances of the same variable name without requiring the complexity involved in allocating new locations.

4. Procedures

A practical language must provide procedures for abstraction purposes, and recursion. In Sections 4.1–4.5 we provide semantics for two kinds of parameters: by-reference (**ref**) and by-value (**val**). Other procedure and parameter mechanisms are discussed in Section 4.6. We discuss functions (procedures that return a value) in Section 5.

4.1. Syntax

We extend the language with procedure declarations and calls.

$$c ::= \ldots \mid p(\vec{y}, \vec{e}) \mid (\textbf{procdecl}\ \rho \bullet c)$$

A procedure call is of the form $p(\vec{y}, \vec{e})$, where $p \in \mathit{Ident}$ is a procedure identifier, $\vec{y}$ is a list of variables which are the actual by-reference parameters, and $\vec{e}$ is a list of expressions which are the actual by-value parameters.

A command $(\textbf{procdecl}\ \rho \bullet c)$, where ρ is a *State*, states that c may use the procedures declared in ρ. Because ρ has the type *State*, there is semantically no difference to $(\textbf{state}\ \rho \bullet c)$, and therefore we define

$$(\textbf{procdecl}\ \rho \bullet c) \mathrel{\widehat{=}} (\textbf{state}\ \rho \bullet c).$$

However, to aid readability we distinguish the two, with the convention that a procedure declaration ρ defines procedures only, that is, the range of ρ are special elements of *Val* called *procedure denotations*, described below. Furthermore, a procedure declaration is constant.

As with states, procedure declarations can be nested, and subprocesses can define their own local version of procedures. This abstract syntax corresponds to concrete syntax of a new block containing c which declares the procedures in $\mathrm{dom}(\rho)$ as local to c.

4.2. Procedure denotations

A procedure may take as arguments a list of by-reference parameters and a list of by-value parameters. As such, a procedure denotation is a function from two lists to a command, the body of the procedure. The body of the procedure is contained within two scoping constructs: the formal by-reference parameters are *dynamically renamed* to the corresponding actual parameters, and the actual by-value parameters are used to initialise the corresponding formal by-value parameters.

For example, consider the following concrete syntax that declares a procedure *square* that updates its by-reference parameter, z, with the square of the value of its by-value parameter, x.

$$\mathit{square}(\textbf{ref}\ z, \textbf{val}\ x) \ \mathrel{\widehat{=}}\ z := x^2$$

This is syntactic sugar representing the mapping of the procedure identifier *square* to the following procedure denotation, $\mathsf{sq}_{\mathsf{den}}$.

$$\mathsf{sq}_{\mathsf{den}} \mathrel{\widehat=} (\lambda\, U \bullet \lambda\, V \bullet (\mathbf{rn}\ \{z \mapsto U\} \bullet (\mathbf{state}\ \{x \mapsto V\} \bullet z := x^2))) \tag{14}$$

It is a function which accepts the metavariable U of type *Ident*, representing the actual by-reference parameter, and the metavariable V of type *Val*, representing the actual by-value parameter. The formal by-reference parameter z is *dynamically renamed* (**rn**) to the identifier passed via U (the semantics is explained in Section 4.3), and the formal by-value parameter x is initialised to the value passed via V. The body of the procedure remains the same.

A procedure declaration that defines *square* as accessible by program c is written

$$(\mathbf{procdecl}\ \{\mathit{square} \mapsto \mathsf{sq}_{\mathsf{den}}\} \bullet c).$$

More generally, we allow procedure denotations to accept multiple by-reference and by-value parameters, which are passed in two lists, respectively.

$$\mathit{ProcDen} \mathrel{\widehat=} \mathbf{list}\ \mathit{Ident} \nrightarrow (\mathbf{list}\ \mathit{Val} \nrightarrow \mathit{Cmd})$$

Elements of type *ProcDen* are partial functions because they are defined only for the correct number of actual parameters; in a well-formed program the application of a *ProcDen* will always be defined. Although technically lists are the parameters to a procedure denotation, when dealing with particular definitions we write (actual and formal) parameters as comma-separated lists of expressions. When there are no by-value parameters or no by-reference parameters, we omit the corresponding empty list of parameters. In the abstract syntax the by-reference parameters precede the by-value, although this need not be the case in the concrete syntax.

Therefore, the general form of a procedure declaration in pseudo-code,

$$p(\mathbf{ref}\ \vec{z}, \mathbf{val}\ \vec{x}) \mathrel{\widehat=} c,$$

where the formals $\vec{z}$ and $\vec{x}$ are lists of distinct identifiers that do not overlap, is represented in our language by a mapping from procedure identifier p to

$$(\lambda\, \vec{U} \bullet \lambda\, \vec{V} \bullet\ \mathbf{rn}\ (\vec{z} \mapsto \vec{U}) \bullet (\mathbf{state}\ (\vec{x} \mapsto \vec{V}) \bullet c)). \tag{15}$$

The syntax $(\vec{a} \mapsto \vec{b})$, defined if $\vec{a}$ and $\vec{b}$ are of equal length, is the mapping formed by pairing corresponding parameters in each list. The lists $\vec{U}$ and $\vec{V}$ are placeholders for the actual by-reference and by-value parameters, respectively (note that it may be assumed that the names of such metavariables do not clash with elements of *Ident* appearing in the abstract syntax). A local state mapping the formal by-value parameters $\vec{x}$ to $\vec{V}$ is created as a context for the body of the procedure c. In addition, any context-dependent behaviour of c which contains the formal by-reference parameters is renamed to contain the actual by-reference parameters. To be well-formed, the lengths of $\vec{z}$ and $\vec{U}$, and of $\vec{x}$ and $\vec{V}$, must be the same, i.e., the number of actuals must match the number of formals.

4.3. Renaming

We extend the set of commands to include dynamic renaming,

$$c ::= \ldots \mid (\mathbf{rn}\ \Theta \bullet c)$$

where Θ is a partial function on identifiers, $\mathit{Ident} \nrightarrow \mathit{Ident}$. A command $(\mathbf{rn}\ \{x \mapsto y\} \bullet c)$ describes local aliasing, where the effect of command c on variable x is transformed so that c affects some other variable y instead. This is precisely the case for by-reference procedure parameters, where the effect on a formal parameter becomes an effect on the corresponding actual parameter. The rules for renaming are presented in Fig. 9. Rule 18(a) states that the renaming is applied to any label, and Rule 18(b) states that a renaming terminates when its command terminates.

The label $\ell\Theta$ in Rule 18(a) is obtained by a straightforward replacement of variables in the domain of Θ with their new names in the range. The renaming has no effect on variables outside its domain.

$$\ell\Theta = \begin{cases} \theta(x) = v & \text{if } \ell \text{ is } x = v \text{ and } x \in \mathrm{dom}(\Theta) \\ \theta(x) := v & \text{if } \ell \text{ is } x := v \text{ and } x \in \mathrm{dom}(\Theta) \\ \ell & \text{otherwise} \end{cases} \tag{16}$$

Assume $\Theta \in (Ident \nrightarrow Ident)$ $\vec{y} \in \mathbf{list}\ Ident$ $\vec{e} \in \mathbf{list}\ Expr$ $\vec{v} \in \mathbf{list}\ Val$

Rule 18 (Rename).

$$(a)\ \frac{c \xrightarrow{\ell} c'}{(\mathbf{rn}\ \Theta \bullet c) \xrightarrow{\ell\Theta} (\mathbf{rn}\ \Theta \bullet c')} \qquad (b)\ \ (\mathbf{rn}\ \Theta \bullet \mathbf{nil}) \longrightarrow \mathbf{nil}$$

Rule 19 (Procedure call).

$$(a)\ \frac{\vec{e} \xrightarrow{\ell} \vec{e}'}{p(\vec{y}, \vec{e}) \xrightarrow{\ell} p(\vec{y}, \vec{e}')}$$

$$(b)\ \frac{\mathsf{p}_{\mathsf{den}} = (\lambda\, \vec{U} \bullet \lambda\, \vec{V} \bullet (\mathbf{rn}\ (\vec{z} \mapsto \vec{U}) \bullet (\mathbf{state}\ (\vec{x} \mapsto \vec{V}) \bullet c)))}{p(\vec{y}, \vec{v}) \xrightarrow{p=\mathsf{p}_{\mathsf{den}}} (\mathbf{rn}\ (\vec{z} \mapsto \vec{y}) \bullet (\mathbf{state}\ (\vec{x} \mapsto \vec{v}) \bullet c))}$$

Fig. 9. Rules for procedure call and renaming.

For example:

$$(x = v)\{x \mapsto y\} = (y = v) \qquad (x := v)\{x \mapsto y\} = (y := v)$$

Hence the context-dependent behaviour of $(\mathbf{rn}\ \Theta \bullet c)$ is the context-dependent behaviour of c, but with each variable $x \in \mathrm{dom}(\Theta)$ replaced with $\Theta(x)$. This corresponds to our intuition about procedure calls, which is that the effect (or constraints) on the by-reference formal parameters is mapped to an equivalent effect on the actuals. Since the labels specify the variable being tested or modified, as shown above it is straightforward to map behaviour from one variable (x) onto another (y). As a consequence, this also helps to simplify the semantics of recursion, including mutual recursion, and concurrent processes calling the same procedure in parallel. Plotkin's explicit aliasing construct '$x == y;\ c$', which introduces a new local name x as an alias for the variable y within c, corresponds to the renaming command $(\mathbf{rn}\ \{x \mapsto y\} \bullet c)$. In Plotkin's semantics, the alias is elaborated into an environment (a mapping from x to the location of y), which then forms part of the explicit context in the transition rules; in our semantics, the renaming command forms part of the implicit context. Plotkin defines the semantics of by-reference parameters via the alias construct [33, Sections 3.4 and 4.3], and likewise we use renaming to define by-reference parameters.

4.4. Procedure call

Our general approach is that procedure bodies replace procedure calls at the point of the call (the *Copy Rule* from *ALGOL-60* [3]). For simplicity we first present a *dynamic binding* [21] strategy for global variables appearing in procedure bodies, that is, free identifiers in procedure bodies refer to the closest defining occurrence at run time. We discuss how to achieve *static binding* in procedure bodies in Section 9.1.

The rules for a procedure call are given in Fig. 9. Rule 19(a) states that a procedure call $p(\vec{y}, \vec{e})$ first evaluates its by-value actual parameters, $\vec{e}$. The evaluation of a list of expressions is left-to-right—its formal definition is straightforward and deferred until later (Rule 20). Rule 19(b) states that when the evaluation of the by-value parameters is finished, p is evaluated in context to its procedure denotation $\mathsf{p}_{\mathsf{den}}$. Given that $\mathsf{p}_{\mathsf{den}}$ is of the standard form (15), the procedure call is replaced by an instance of the procedure body c, contained in a context in which the formal by-reference parameters $\vec{z}$ are dynamically renamed to the actual by-reference parameters $\vec{y}$, and the values $\vec{v}$ initialise the formal by-value parameters $\vec{x}$. Note that Rule 19(b) is similar to Rule 1: it may be rewritten, somewhat awkwardly, so that the premise is removed and the standard form for a procedure denotation appears in place of $\mathsf{p}_{\mathsf{den}}$ in the transition in the conclusion of the rule. However, for well-formed commands we may assume that procedure identifiers such as p are always mapped to a $\mathsf{p}_{\mathsf{den}}$ of the standard form, and hence a more compact version of the rule may be given.

$$p(\vec{y}, \vec{v}) \xrightarrow{p=\mathsf{p}_{\mathsf{den}}} \mathsf{p}_{\mathsf{den}}(\vec{y})(\vec{v}) \tag{17}$$

For brevity, in further variants of the procedure call rule we will use this version, where it is understood the application $\mathsf{p}_{\mathsf{den}}(\vec{y})(\vec{v})$ may be β-reduced.

4.5. Examples

In this section we provide examples of procedure calls in our framework, including recursion. To simplify the presentation we use the following equivalences.

$$(\mathbf{state}\ \{x \mapsto v\} \bullet c) \approx c \quad \text{provided } x \text{ nfi } c, c \text{ does not contain procedure calls} \tag{18}$$

$$(\mathbf{rn}\ \{x \mapsto x\} \bullet c) \approx c \tag{19}$$

The notation $c \approx d$ means that c and d are *trace equivalent*, that is, ignoring internal steps, they generate the same traces. Hence, to simplify the presentation, we replace occurrences of the left-hand sides with the right-hand sides. The proviso on (18) requires that the local variable x does not appear free in c (abbreviated as x nfi c), and that c does not contain procedure calls (since such calls may unfold to procedure bodies that refer to x). Intuitively, the equivalence holds because by assumption any context-dependent behaviour ℓ exhibited by c can never reference x, and hence the state has no effect on ℓ (see Definition 9). Equivalence (19) holds because the renaming is an identity function, and hence has no effect on any context-dependent behaviour of c from (16). We discuss command equivalence in more detail in Section 8.

4.5.1. Standard procedure call

Consider a program for calculating the square of 3 and storing it in a global variable a, using the procedure *square* from (14),

$$(\mathbf{procdecl}\ \{square \mapsto \mathsf{sq}_{\mathsf{den}}\} \bullet square(a, 3)). \tag{20}$$

From Rule 19(b) and Rule 11(a) the procedure call is replaced by the application of the actual parameters to the procedure denotation for *square*, i.e., $\mathsf{sq}_{\mathsf{den}}(a)(3)$, which gives the following command.

$$(\mathbf{procdecl}\ \{square \mapsto \mathsf{sq}_{\mathsf{den}}\} \bullet (\mathbf{rn}\ \{z \mapsto a\} \bullet (\mathbf{state}\ \{x \mapsto 3\} \bullet z := x^2)))$$

The assignment expression, x^2, is evaluated to 9 using the evaluation rules. This eliminates references to x and we omit the state from the presentation using (18). The renaming step of the program immediately after the evaluation is shown below.

$$\frac{z := 9 \xrightarrow{z:=9} \mathbf{nil}}{(\mathbf{rn}\ \{z \mapsto a\} \bullet z := 9) \xrightarrow{a:=9} (\mathbf{rn}\ \{z \mapsto a\} \bullet \mathbf{nil})} \tag{21}$$

The transition is justified by Rule 4(b) and Rule 18(a). After renaming, the context-dependent behaviour is updating a to 9, as expected.

To summarise:

$$\begin{array}{rl}
 & square(a, 3) \\
\xrightarrow{square=\mathsf{sq}_{\mathsf{den}}} & \text{Rule 19(b)} \\
 & \mathsf{sq}_{\mathsf{den}}(a)(3) \\
= & (14) \\
 & (\mathbf{rn}\ \{z \mapsto a\} \bullet (\mathbf{state}\ \{x \mapsto 3\} \bullet z := x^2)) \\
\Longrightarrow & \text{Expression evaluation, (18)} \\
 & (\mathbf{rn}\ \{z \mapsto a\} \bullet z := 9) \\
\xrightarrow{a:=9} & \text{From (21)} \\
 & (\mathbf{rn}\ \{z \mapsto a\} \bullet \mathbf{nil}) \\
\longrightarrow & \text{Rule 18(b)} \\
 & \mathbf{nil}
\end{array}$$

The label on the first step is hidden outside the scope of (20). Therefore, the effect of this execution is updating a to 9. Note that there was no need to allocate locations for x or z.

4.5.2. Resolving name clashes

In this section we demonstrate how name clashes that result from procedure calls are resolved in our framework.

Case 1. Firstly, consider the case where a call to *square* occurs in a context which already contains the variable x, i.e., a variable with the same name as the formal by-value parameter. We implicitly assume the declaration of *square* as in (20).

$$(\mathbf{state}\ \{x \mapsto 5\} \bullet square(a, x - 2))$$

The procedure call first evaluates its by-value parameter, $x - 2$, to 3, in the current context in which x is 5.

$$(\mathbf{state}\ \{x \mapsto 5\} \bullet \mathit{square}(a, 3))$$

The call itself is then expanded as follows.

$$(\mathbf{state}\ \{x \mapsto 5\} \bullet (\mathbf{rn}\ \{z \mapsto a\} \bullet (\mathbf{state}\ \{x \mapsto 3\} \bullet z := x^2)))$$

There is a new inner state which maps x to the value 3. However, any computation within the call that refers to x will be limited to updating only the innermost x, since Rule 12(a) hides the effect on x in the label. An alternative approach, as mentioned in the ALGOL-60 report [3], is to resolve name clashes between actual parameters and local procedure variables by renaming the latter. To formally specify this approach requires globally maintaining the set of used variable names; in our setting clashes are resolved by keeping the name spaces separate through renamings.

Case 2. Another potential name clash occurs when x is passed as the actual by-reference parameter.

$$(\mathbf{state}\ \{x \mapsto 5\} \bullet \mathit{square}(x, 3))$$

After expanding the call, we have

$$(\mathbf{state}\ \{x \mapsto 5\} \bullet (\mathbf{rn}\ \{z \mapsto x\} \bullet (\mathbf{state}\ \{x \mapsto 3\} \bullet z := x^2)))$$

The expression x^2 is evaluated with respect to the innermost x, that is, to 9. The context-dependent behaviour of the assignment becomes $z := 9$, which is unaffected by the inner state, and is renamed to $x := 9$ by the renaming. This updates the outermost x to 9, as expected.

Case 3. If x appears as a non-local variable in the body of a procedure, and is also passed as an actual by-reference parameter to that procedure, then aliasing occurs. However, the renaming will have no effect on the non-local occurrences of x, and the procedure body will execute as expected. For instance, the following procedure increases the value of its by-reference parameter by the value of a non-local variable a.

$$p(\mathbf{ref}\ z) \mathrel{\widehat{=}} z := a + z$$

A call $p(a)$ expands as follows.

$$\mathbf{rn}\ \{z \mapsto a\} \bullet z := a + z$$

The evaluation of the assignment first looks for the value of a in context, and then the value of z, which is renamed back to a. The assignment itself therefore has the effect of doubling the value of a (assuming no other concurrent process modifies a). For example, assuming $a = 1$ in context, the trace of the assignment is $(a = 1).(z = 1).(z := 2)$, while at the outer level the trace becomes $(a = 1).(a = 1).(a := 2)$.

Case 4. Aliasing can occur if x is passed as the actual parameter to two by-reference parameters. This will result in a renaming $\Theta = \{z_1 \mapsto x, z_2 \mapsto x\}$, which has a similar effect to case 3 above.

4.5.3. Recursion

Consider the following program for finding the summation from 1 to some number a, and storing the (successive) results in b. It is a recursive version of (12).

$$\begin{array}{l} \mathit{sum}(\mathbf{ref}\ b, \mathbf{val}\ a) \mathrel{\widehat{=}} \\ \quad \mathbf{if}\ a = 0\ \mathbf{then}\ b := 0 \\ \quad \mathbf{else}\ (\mathit{sum}(b, a - 1)\ ;\ b := a + b) \end{array} \tag{22}$$

Call the corresponding procedure denotation $\mathsf{sum}_{\mathsf{den}}$. We use this to calculate the sum to 2 and store the result in x.

$$(\mathbf{procdecl}\ \{\mathit{sum} \mapsto \mathsf{sum}_{\mathsf{den}}\} \bullet \mathit{sum}(x, 2)) \tag{23}$$

Applying Rule 19(b) the procedure call expands to the following.

$$\begin{array}{l} \mathbf{rn}\ \{b \mapsto x\} \bullet \mathbf{state}\ \{a \mapsto 2\} \bullet \\ \quad \mathbf{if}\ a = 0\ \mathbf{then}\ b := 0 \\ \quad \mathbf{else}\ (\mathit{sum}(b, a - 1)\ ;\ b := a + b) \end{array}$$

By expression evaluation and Rule 7(a) and Rule 7(c) we select the second branch of the conditional.

$$\begin{array}{l} \mathbf{rn}\ \{b \mapsto x\} \bullet \mathbf{state}\ \{a \mapsto 2\} \bullet \\ \quad (\mathit{sum}(b, a - 1)\ ;\ b := a + b) \end{array} \tag{24}$$

We expand the recursive call in the same manner as the original call, as (24) is still in the context as in (23). Note we get nested renamings, as well as nested states.

$$
\begin{array}{l}
\textbf{rn}\ \{b \mapsto x\} \bullet \textbf{state}\ \{a \mapsto 2\} \bullet \\
\quad (\textbf{rn}\ \{b \mapsto b\} \bullet \textbf{state}\ \{a \mapsto 1\} \bullet \\
\qquad \textbf{if}\ a = 0\ \textbf{then}\ b := 0 \\
\qquad \textbf{else}\ (sum(b, a - 1)\ ;\ b := a + b))\ ; \\
\quad b := a + b
\end{array}
$$

The inner renaming of b to b has no effect (this is because the recursive calls all refer to the same by-reference variable). For presentation purposes we omit the renaming by (19). By Rule 7 we select the second branch of the conditional again.

$$
\begin{array}{l}
\textbf{rn}\ \{b \mapsto x\} \bullet \textbf{state}\ \{a \mapsto 2\} \bullet \\
\quad (\textbf{state}\ \{a \mapsto 1\} \bullet \\
\qquad (sum(b, a - 1)\ ;\ b := a + b))\ ; \\
\quad b := a + b
\end{array}
$$

Note there are nested names a. The inner a has the value 1 (the innermost recursive call), while the outer a has the value 2.

We expand the call to *sum* again, with 0 as the actual value for a, which means by Rule 7(b) we select the first branch of the conditional.

$$
\begin{array}{l}
\textbf{rn}\ \{b \mapsto x\} \bullet \textbf{state}\ \{a \mapsto 2\} \bullet \\
\quad (\textbf{state}\ \{a \mapsto 1\} \bullet \\
\qquad (\textbf{state}\ \{a \mapsto 0\} \bullet \\
\qquad\quad b := 0)\ ; \\
\qquad b := a + b)\ ; \\
\quad b := a + b
\end{array}
\tag{25}
$$

This program now takes a series of straightforward steps which involve expression evaluation, removing **nil**s, and assignments to b. Within the scope of the renaming $\{b \mapsto x\}$, the context-dependent behaviour is the following trace:

$$(b := 0).(b = 0).(b := 1).(b = 1).(b := 3)$$

Outside the renaming, that is, in the context of the original call, the context-dependent behaviour is mapped onto x, giving

$$(x := 0).(x = 0).(x := 1).(x = 1).(x := 3) \tag{26}$$

Since x is the actual by-reference parameter, it is updated and tested during execution, with the final value of 3 as expected.

In the case of a non-terminating recursion, our rules will result in an infinite sequence of transitions.

4.6. Other types of parameters

4.6.1. Call by name

Call-by-name can be viewed as a generalisation of call by reference, where instead of variables, unevaluated expressions are passed as actual parameters. Under this interpretation, renamings and labels must also be generalised to handle expressions in place of variables. Therefore, operations on the formal by-name parameters are renamed so that they become operations on the actual by-name parameters (expressions). When the behaviour is expression evaluation, the renaming results in straightforward behaviour. If the actual parameters are assigned to, the language must be rich enough to handle expressions on the left-hand side of assignments. This is discussed in Section 9.2.

4.6.2. Call by result

Given a procedure p with a *by-result* parameter z, a call $p(a)$ is designed so that at the end of the call the final value of z is copied back to a [10]. The variable a will be otherwise unchanged during the execution of p (unless a occurs free in the body of p, or a is also passed as a by-reference parameter). The by-result mechanism avoids the issue of incremental updates of a by-reference parameter, such as in (26).

A by-result parameter may be treated as a special case of a by-reference parameter, as follows: if z is the name of a formal by-result parameter, then the procedure accepts a by-reference parameter with a fresh name, say z_r, while z becomes a local variable of the procedure body. The only use of z_r is an assignment $z_r := z$ at the end of the procedure body.

If a variable, a, is used as an actual by-result parameter more than once in a single call, the order of the final assignments may affect the final value of a. Several options exist for handling this case; for instance, Gries [10, Section 12.1] assigns the final values in a strict left-to-right order according the formal parameter list, and this is easily modelled in our framework.

The extension of by-result to by-value-result is straightforward, by using the evaluation of the actual by-result parameter to initialise z.

5. Complex expressions

In the main section of the paper we have assumed a basic expression language. However programming languages often allow function calls to appear in expressions, either in assignments or as parameters to other function or procedure calls. We explore this option in this section, and show how we allow small-step function evaluation within expressions.

5.1. Syntax

We extend the syntax of expressions below.

$$e ::= v \mid x \mid \mathsf{op}(\vec{e}) \mid f(\vec{y}, \vec{e}) \mid \mathsf{fneval}(c)$$

An expression may be a value, a variable, some operator op applied to a list of expressions, a function call on f, where f is an identifier, or a function body, written $\mathsf{fneval}(c)$, where c is a command.

The $\mathsf{op}(\vec{e})$ syntax generalises the basic expressions we gave earlier, which allowed only binary addition. Operators are distinct from functions (below)—operators are expression-level objects that are primitive to the language, while functions are command-level objects.

A function call $f(\vec{y}, \vec{e})$ is handled like a procedure call (including by-reference parameters), except that the call is evaluated to a $\mathsf{fneval}(c)$ expression, where c is the instantiated body of f. A $\mathsf{fneval}(c)$ expression is an expression whose value is the value returned by the command c. A value is returned from a function by a **return** e command, where e is an expression. To be well-formed, a function body must always return a value using a **return** command; apart from this restriction, a function denotation is just a procedure denotation.

5.2. Semantics

The expression evaluation rules are given in Fig. 10. Rule 20 states that a list of expressions is evaluated from left-to-right. An operation op on parameters $\vec{e}$ evaluates $\vec{e}$ until it is a ground list of values (Rule 21(a)), then the meaning of op, given by some evaluator *opf*, is used to calculate the resulting value (Rule 21(b)). If an evaluator is undefined for its inputs, then the evaluation is blocked from proceeding (an alternative would be to extend the language to include exceptions to handle the undefined cases). Rule 2 for addition may be derived from Rules 20 and 21, where op is '+' (written in infix form) and *opf* is '+', the semantics of addition (again written in infix form).

A function call on f first evaluates its by-value parameters (Rule 22(a)), and when ground, applies the definition of f, $\mathsf{f}_{\mathsf{den}}$, to the parameters, and wraps the resulting command as an fneval expression (Rule 22(b)). The rule is similar to Rule 19 for procedure calls (which we write using the shorthand in (17)).

An expression $\mathsf{fneval}(c)$ is evaluated through executing c (Rule 23(a)), until c returns a value. In this case, the fneval expression evaluates to the returned value and execution of c finishes (Rule 23(b)).

A command **return** e first evaluates the expression e via Rule 24(a), then exposes the calculated value in the transition label before terminating via Rule 24(b). We therefore extend the type *Label* to include **return** values.

$$\ell ::= \ldots \mid \mathbf{return}\ v$$

The return label is treated specially only by a fneval expression. This means that all of the rules in Figs. 3 and 9 still hold for $\ell \in \mathit{Label}$ using the extended definition. Rules 13 and 14 also hold, using Definition 9: a label **return** v is not affected by, nor affects, a local state, and is always consistent with a local state. A renaming has no effect on a return label, as given by (16).

5.3. Example

We rewrite the *sum* procedure from (22) as a recursive function. Let $\mathsf{sumf}_{\mathsf{den}}$ be the following function definition of *sum* (it is an instance of (15) with no by-reference parameters).

$$\begin{array}{l}(\lambda\, V \bullet \mathbf{state}\ \{a \mapsto V\} \bullet \\ \qquad \mathbf{if}\ a = 0\ \mathbf{then\ return}\ 0 \\ \qquad \mathbf{else\ return}\ \mathit{sum}(a-1) + a)\end{array}$$

First consider executing the following command.

$$(\mathbf{procdecl}\ \{\mathit{sum} \mapsto \mathsf{sumf}_{\mathsf{den}}\} \bullet x := \mathit{sum}(0))$$

Rule 20 (**Evaluate expression list).**

$$(a)\ \frac{e \xrightarrow{\ell} e'}{(e.\vec{e}) \xrightarrow{\ell} (e'.\vec{e})} \qquad (b)\ \frac{\vec{e} \xrightarrow{\ell} \vec{e}'}{(v.\vec{e}) \xrightarrow{\ell} (v.\vec{e}')}$$

Rule 21 (**Evaluate general operator).**

$$(a)\ \frac{\vec{e} \xrightarrow{\ell} \vec{e}'}{\mathsf{op}(\vec{e}) \xrightarrow{\ell} \mathsf{op}(\vec{e}')} \qquad (b)\ \frac{\vec{v} \in \mathrm{dom}(opf)}{\mathsf{op}(\vec{v}) \longrightarrow opf(\vec{v})}$$

Rule 22 (**Evaluate function call).**

$$(a)\ \frac{\vec{e} \xrightarrow{\ell} \vec{e}'}{f(\vec{y}, \vec{e}) \xrightarrow{\ell} f(\vec{y}, \vec{e}')} \qquad (b)\ f(\vec{y}, \vec{v}) \xrightarrow{f=\mathsf{f}_{\mathsf{den}}} \mathsf{fneval}(\mathsf{f}_{\mathsf{den}}(\vec{y})(\vec{v}))$$

Rule 23 (**Evaluate** fneval**).**

$$(a)\ \frac{c \xrightarrow{\ell} c' \quad (\forall v \bullet \ell \neq \mathbf{return}\ v)}{\mathsf{fneval}(c) \xrightarrow{\ell} \mathsf{fneval}(c')} \qquad (b)\ \frac{c \xrightarrow{\mathbf{return}\ v} c'}{\mathsf{fneval}(c) \longrightarrow v}$$

Rule 24 (**Return).**

$$(a)\ \frac{e \xrightarrow{\ell} e'}{\mathbf{return}\ e \xrightarrow{\ell} \mathbf{return}\ e'} \qquad (b)\ \mathbf{return}\ v \xrightarrow{\mathbf{return}\ v} \mathbf{nil}$$

Fig. 10. Small-step complex expression evaluation.

The assignment evaluates as follows, from Rule 22.

$$x := \mathsf{fneval}(\mathbf{state}\ \{a \mapsto 0\} \bullet\ \mathbf{if}\ a = 0\ \mathbf{then\ return}\ 0\ \mathbf{else\ return}\ sum(a-1) + a)$$

From expression evaluation and Rule 7(b), the first branch of the conditional is selected.

$$x := \mathsf{fneval}(\mathbf{state}\ \{a \mapsto 0\} \bullet \mathbf{return}\ 0)$$

We eliminate the now unnecessary state containing a using (18). Then the execution terminates and the value 0 is returned.

$$\dfrac{\dfrac{\mathbf{return}\ 0 \xrightarrow{\mathbf{return}\ 0} \mathbf{nil}}{\mathsf{fneval}(\mathbf{return}\ 0) \longrightarrow 0}}{x := \mathsf{fneval}(\mathbf{return}\ 0) \longrightarrow x := 0}$$

Rule 24(*b*)

Rule 23(*b*)

Rule 4(*a*)

Now consider executing the following program.

$$(\mathbf{procdecl}\ \{sum \mapsto \mathsf{sumf}_{\mathsf{den}}\} \bullet x := sum(1))$$

Following similar reasoning as above, the assignment simplifies to

$$x := \mathsf{fneval}(\mathbf{state}\ \{a \mapsto 1\} \bullet \mathbf{return}(sum(0) + a))$$

Now the evaluation of the expression $sum(0)$ also continues as above, resulting in a nested fneval expression (eliminating again an unnecessary declaration of a).

$$x := \mathsf{fneval}(\mathbf{state}\ \{a \mapsto 1\} \bullet (\mathbf{return}(\mathsf{fneval}(\mathbf{return}\ 0) + a)))$$

The inner fneval evaluates to 0, as described above.

$$x := \mathsf{fneval}(\mathbf{state}\ \{a \mapsto 1\} \bullet (\mathbf{return}(0 + a)))$$

The expression $0 + a$ evaluates to 1, and the assignment overall evaluates to $x := 1$.

A similar transformation can be used to straightforwardly execute $x := sum(2)$, which generates the trace $x := 3$. Note that this is in contrast to earlier calculations of the sum which resulted in successive updates to x.

6. Atomicity

In this section we introduce a command for atomic execution, which removes the possibility of interference during execution. Such a command may be useful for prohibiting unwanted interleavings. For instance, consider the following program that performs two concurrent increments of y before assigning the result to x.

$$(\mathbf{state}\ \{y \mapsto 0\} \bullet (y := y + 1 \parallel y := y + 1)\ ;\ x := y) \tag{27}$$

Naively, it would appear that there is exactly one trace of this program, $(x := 2)$, since y is incremented twice. That is, we may expect the following two sub-executions to appear in the trace of (27).

$$y := y + 1 \xrightarrow{(y=0).(y:=1)} \mathbf{nil} \qquad y := y + 1 \xrightarrow{(y=1).(y:=2)} \mathbf{nil} \tag{28}$$

It is certainly possible for the execution of the two assignments to occur sequentially like this. However, because expression evaluation is non-atomic, both expressions $y + 1$ may evaluate to 1, and hence the final value of x may be 1. This is because the executions may interleave, and both may read the initial value 0 for y before either assigns to y, and hence in both cases the assignment expression will evaluate to 1.

$$(y := y + 1) \parallel (y := y + 1) \xrightarrow{(y=0).(y=0)} (y := 1) \parallel (y := 1) \xrightarrow{(y:=1).(y:=1)} \mathbf{nil} \tag{29}$$

To avoid this problem of interference within the execution of a command, but allowing interleaving between commands, we define an atomic command, (**atomic** c). Rule 25 in Fig. 11 states that an atomic command (**atomic** c) executes with context-dependent behaviour given by the trace ℓs if c may execute until normal completion with a consistent trace ℓs, and ℓs does not abruptly terminate due to a return from a function call. The latter condition is given by $\mathbf{return} \notin \ell s$, which we use as an abbreviation for $\neg\ (\exists v{:}\ Val \bullet \mathbf{return}\ v \in \mathrm{ran}(\ell s))$, i.e., the trace ℓs does not contain a (**return** v) label, for any v ($\mathrm{ran}(\ell s)$ gives the set of elements in ℓs). The predicate $consistent(\ell s)$ ensures that ℓs represents the execution of c without interference, and is defined formally below using a lifted version of *consistent* from Definition 9. Note that Rule 25 does not allow partial computations of c, nor abrupt termination of c, and it does not allow c to fail to terminate.

Rule 25 (**Atomic**).

$$\frac{c \xrightarrow{\ell s} \mathbf{nil} \quad \mathbf{return} \notin \ell s \quad consistent(\ell s)}{(\mathbf{atomic}\ c) \xrightarrow{\ell s} \mathbf{nil}}$$

Rule 26 (**Atomic with return**).

$$\frac{c \xrightarrow{\ell s.(\mathbf{return}\ v)} c' \quad \mathbf{return} \notin \ell s \quad consistent(\ell s)}{(\mathbf{atomic}\ c) \xrightarrow{\ell s.(\mathbf{return}\ v)} \mathbf{nil}}$$

Rule 27 (**Atomic** fneval).

$$\frac{c \xrightarrow{\ell s.(\mathbf{return}\ v)} c'}{\mathsf{fneval}(c) \xrightarrow{\ell s} v}$$

Rule 28 (**Let expressions for traces**).

$$\frac{e \xrightarrow{\ell s} e' \quad consistent(\ell s, \sigma)}{(\mathbf{let}\ \sigma\ \mathbf{in}\ e) \xrightarrow{\ell s[\sigma]} (\mathbf{let}\ \sigma[\ell s]\ \mathbf{in}\ e')}$$

Rule 29 (**Local state for traces**).

$$\frac{c \xrightarrow{\ell s} c' \quad consistent(\ell s, \sigma)}{(\mathbf{state}\ \sigma \bullet c) \xrightarrow{\ell s[\sigma]} (\mathbf{state}\ \sigma[\ell s] \bullet c')}$$

Fig. 11. Rules for atomicity.

The case where the execution of c abruptly terminates due to a **return** command is given by Rule 26. This rule states that the first time a **return** command is executed in c, the atomic command terminates, even if there are further commands in c (this may be the situation if, for instance, a loop contains a **return** command to handle a special case). Rule 27 handles the case where a function call contains an atomic block that terminates with a **return**. It is a generalisation of Rule 23(b); we also require a straightforward generalisation of Rule 23(a) to handle the case where an atomic trace does not contain a **return** label. Note that by Rules 25 and 26, any trace generated by an atomic command may contain at most one **return** label, and any such label must be the last. Hence, we do not need to check the condition **return** $\notin \ell s$ in Rule 27.

The **atomic** command generates a trace as the annotation on the transition arrow, rather than a single label as with the other commands. Hence we extend the label type in Fig. 3 to range over traces, i.e.,

$$\ell ::= \ldots \mid \ell s$$

The rules from Fig. 3 still apply with this extended label type, if we allow rules that are valid for τ labels be valid also for the empty trace ϵ or a trace of purely internal steps. We also define a trace version of Rule 18 for renamings where $\ell s\Theta$ is the list resulting from applying the renaming Θ to each label in ℓs according to (16). The only additional rules required are for **let** expressions and local states to handle traces instead of a single label—see Rules 28 and 29, which are Rules 13 and 14 lifted to traces, respectively. The functions in Definition 9 are straightforwardly lifted to traces as follows.

$$\begin{aligned} \epsilon[\sigma] &= \epsilon & \sigma[\epsilon] &= \sigma \\ \ell.\ell s[\sigma] &= (\ell[\sigma]).(\ell s[\sigma]) & \sigma[\ell.\ell s] &= (\sigma[\ell])[\ell s] \\ \mathit{consistent}(\epsilon, \sigma) &\Leftrightarrow \mathit{true} \\ \mathit{consistent}(\ell.\ell s, \sigma) &\Leftrightarrow \mathit{consistent}(\ell, \sigma) \wedge \mathit{consistent}(\ell s, \sigma[\ell]) \end{aligned} \tag{30}$$

Hence, $\ell s[\sigma]$ is the trace ℓs with all labels relevant to the local state replaced by internal steps (τ), $\sigma[\ell s]$ is σ updated according to the trace with the updates occurring in the order they appear in ℓs, and a trace is *consistent* if its first label is consistent, and the rest of the trace is consistent with the new state. Therefore, Rule 29 is defined as if the labels in ℓs are successively executed using Rule 14. For instance, if $c \xrightarrow{(x:=1).(y=1).(z:=1)} c'$ then

$$(\textbf{state } \{x \mapsto 0\} \bullet c) \xrightarrow{(y=1).(z:=1)} (\textbf{state } \{x \mapsto 1\} \bullet c')$$

We may define $\mathit{consistent}(\ell s)$, as used in Rule 25, using the definition of $\mathit{consistent}(\ell s, \sigma)$ above.

$$\mathit{consistent}(\ell s) \mathrel{\widehat{=}} (\exists \sigma : \mathit{vars}(\ell s) \to \mathit{Val} \bullet \mathit{consistent}(\ell s, \sigma))$$

The set $\mathit{vars}(\ell s)$ contains all of the variables mentioned in the trace ℓs, and hence ℓs is consistent if there exists some mapping σ of the variables in ℓs with which it is consistent.

As an example, consider the following variant of (27), in which the assignments to y are defined to work atomically, that is, interference during execution is prevented.

$$(\textbf{state } \{y \mapsto 0\} \bullet ((\textbf{atomic } y := y + 1) \parallel (\textbf{atomic } y := y + 1)) \,;\, x := y) \tag{31}$$

We first note that, for any integer v,

$$(\textbf{atomic } y := y + 1) \xrightarrow{(y=v).(y:=v+1)} \textbf{nil} \tag{32}$$

By Rule 25 no other labels can appear between the read and update of y in the trace (the atomic command binds them together). Hence, both traces in (28) are valid, but the interleaving that occurred in (29) is not. An execution of (31) is given below. The first and second steps use the two executions from (28), respectively, both of which are instances of (32).

$(\textbf{state } \{y \mapsto 0\} \bullet ((\textbf{atomic } y := y + 1) \parallel (\textbf{atomic } y := y + 1)) \,;\, x := y)$

$\longrightarrow$ Rule 4, (28), Rule 25, Rule 6, Rule 5, Rule 29
$(\textbf{state } \{y \mapsto 1\} \bullet (\textbf{nil} \parallel (\textbf{atomic } y := y + 1)) \,;\, x := y)$

$\longrightarrow$ As above
$(\textbf{state } \{y \mapsto 2\} \bullet (\textbf{nil} \parallel \textbf{nil}) \,;\, x := y)$

$\xrightarrow{x:=2}$ Rule 6, Rule 5, Rule 4, Rule 9
nil

Rule 30 (**Value**).

$$v \xrightarrow{\epsilon}\!\!\!\!\rightarrow v$$

Rule 31 (**Variable**).

$$x \xrightarrow{x=v}\!\!\!\!\rightarrow v$$

Rule 32 (**Addition**).

$$\frac{e_1 \xrightarrow{\ell s_1}\!\!\!\!\rightarrow v_1 \quad e_2 \xrightarrow{\ell s_2}\!\!\!\!\rightarrow v_2}{(e_1 + e_2) \xrightarrow{\ell s_1.\ell s_2}\!\!\!\!\rightarrow v_1 + v_2}$$

Rule 33 (**Nil**).

$$\mathbf{nil} \xrightarrow{\epsilon}\!\!\!\!\rightarrow \mathbf{nil}$$

Rule 34 (**Assignment**).

$$\frac{e \xrightarrow{\ell s}\!\!\!\!\rightarrow v}{(x := e) \xrightarrow{\ell s.(x:=v)}\!\!\!\!\rightarrow \mathbf{nil}}$$

Rule 35 (**Seq. composition**).

$$\frac{c_1 \xrightarrow{\ell s_1}\!\!\!\!\rightarrow \mathbf{nil} \quad c_2 \xrightarrow{\ell s_2}\!\!\!\!\rightarrow \mathbf{nil}}{(c_1 \ ; \ c_2) \xrightarrow{\ell s_1.\ell s_2}\!\!\!\!\rightarrow \mathbf{nil}}$$

Rule 36 (**Conditional**).

$$\frac{b \xrightarrow{\ell s_1}\!\!\!\!\rightarrow \mathit{true} \quad c_1 \xrightarrow{\ell s_2}\!\!\!\!\rightarrow \mathbf{nil}}{(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) \xrightarrow{\ell s_1.\ell s_2}\!\!\!\!\rightarrow \mathbf{nil}} \qquad \frac{b \xrightarrow{\ell s_1}\!\!\!\!\rightarrow \mathit{false} \quad c_2 \xrightarrow{\ell s_2}\!\!\!\!\rightarrow \mathbf{nil}}{(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) \xrightarrow{\ell s_1.\ell s_2}\!\!\!\!\rightarrow \mathbf{nil}}$$

Rule 37 (**While**).

$$\frac{b \xrightarrow{\ell s_1}\!\!\!\!\rightarrow \mathit{true} \quad c \ ; (\mathbf{while}\ b\ \mathbf{do}\ c) \xrightarrow{\ell s_2}\!\!\!\!\rightarrow \mathbf{nil}}{(\mathbf{while}\ b\ \mathbf{do}\ c) \xrightarrow{\ell s_1.\ell s_2}\!\!\!\!\rightarrow \mathbf{nil}} \qquad \frac{b \xrightarrow{\ell s}\!\!\!\!\rightarrow \mathit{false}}{(\mathbf{while}\ b\ \mathbf{do}\ c) \xrightarrow{\ell s}\!\!\!\!\rightarrow \mathbf{nil}}$$

Rule 38 (**Interleave**).

$$\frac{c_1 \xrightarrow{\ell s_1}\!\!\!\!\rightarrow \mathbf{nil} \quad c_2 \xrightarrow{\ell s_2}\!\!\!\!\rightarrow \mathbf{nil} \quad \ell s \in (\ell s_1 \mid \ell s_2)}{(c_1 \parallel c_2) \xrightarrow{\ell s}\!\!\!\!\rightarrow \mathbf{nil}}$$

Rule 39 (**Atomic**).

$$\frac{c \xrightarrow{\ell s}\!\!\!\!\rightarrow \mathbf{nil} \quad \mathit{consistent}(\ell s)}{(\mathbf{atomic}\ c) \xrightarrow{\langle \ell s \rangle}\!\!\!\!\rightarrow \mathbf{nil}}$$

Rule 40 (**Let**).

$$\frac{e \xrightarrow{\ell s}\!\!\!\!\rightarrow v \quad \mathit{consistent}(\ell s, \sigma)}{(\mathbf{let}\ \sigma\ \mathbf{in}\ e) \xrightarrow{\ell s[\sigma]}\!\!\!\!\rightarrow v}$$

Rule 41 (**Local state**).

$$\frac{c \xrightarrow{\ell s}\!\!\!\!\rightarrow \mathbf{nil} \quad \mathit{consistent}(\ell s, \sigma)}{(\mathbf{state}\ \sigma \bullet c) \xrightarrow{\ell s[\sigma]}\!\!\!\!\rightarrow \mathbf{nil}}$$

Rule 42 (**Rename**).

$$\frac{c \xrightarrow{\ell s}\!\!\!\!\rightarrow \mathbf{nil}}{(\mathbf{rn}\ \Theta \bullet c) \xrightarrow{\ell s\Theta}\!\!\!\!\rightarrow \mathbf{nil}}$$

Rule 43 (**Procedure call**).

$$\frac{\vec{e} \xrightarrow{\ell s_1}\!\!\!\!\rightarrow \vec{v} \qquad \mathsf{p}_{\mathsf{den}}(\vec{y})(\vec{v}) \xrightarrow{\ell s_2}\!\!\!\!\rightarrow \mathbf{nil}}{p(\vec{y}, \vec{e}) \xrightarrow{\ell s_1.(p=\mathsf{p}_{\mathsf{den}}).\ell s_2}\!\!\!\!\rightarrow \mathbf{nil}}$$

Fig. 12. Big-step semantics.

The two atomic assignment commands are prevented from interfering with each other (although the order in which they are executed is still nondeterministic by Rule 6—in the above execution we arbitrarily chose to execute the left-hand command first).

7. Big-step semantics

In this section we give a big-step (or natural) semantics for expression evaluation and command execution. The big-step style requires fewer rules, although the rules tend to be more complex. In contrast to Plotkin-style big-step semantics, it is straightforward to define the meaning of concurrency in our setting. The main limitation of big-step semantics is that it says nothing about non-terminating executions, which makes it less suitable for specifying the behaviour of, for instance,

reactive systems. As a further consequence, the handling of abrupt termination, e.g., returning from function calls, becomes problematic, as we discuss in more detail below.

The big-step transition relation is decorated by a trace, rather than a single label as in the small-step semantics.

$$\twoheadrightarrow : \mathit{Trace} \rightarrow \mathbb{P}(\mathit{Cmd} \times \mathit{Cmd})$$

The rules for big-step expression evaluation and command execution are given in Fig. 12. A transition $e \xrightarrow{\ell s}\!\!\!\!\twoheadrightarrow v$ states that e evaluates to v under the series of steps in ℓs, and similarly for $c \xrightarrow{\ell s}\!\!\!\!\twoheadrightarrow \mathbf{nil}$. Rule 30 states that a value evaluates to itself with the empty trace, Rule 31 states that a variable evaluates to a value with a singleton trace (cf. Rule 1), and Rule 32 states that the trace for evaluating an addition is the concatenation of the traces resulting from the evaluation of the operands. The evaluation of a **let** expression is similar to the execution of an **state** command, as described below.

Rule 33 gives the empty trace for the execution of **nil**, Rule 34 states that the trace of an assignment is the trace associated with evaluating the expression followed by the update, and Rule 35 for sequential composition concatenates the traces.

Rule 36 states that the trace of a conditional is formed from either the trace of the evaluation of b to *true* followed by the trace of c_1, or b to *false* and c_2. The semantics of a while loop (Rule 37) is to evaluate the guard to *true* and execute c followed by the loop itself again, or an evaluation of the guard to false. As is common with big-step style semantics, the rules do not say anything about non-terminating executions of the loop [19].

Concurrency (Rule 38) is also straightforward to define, where we let $\ell s_1 \mid \ell s_2$ denote the set of traces that are formed from an order-preserving interleaving of the labels in ℓs_1 and ℓs_2 (there are many such traces). Atomicity (Rule 39) is similar to the small-step rule, except that we make it explicit that the consistent trace generated by c is an indivisible step, that is, the trace on the label of an atomic command is a singleton that contains as its element the trace ℓs. This follows the extension of the label type to include traces in Section 6.

Rule 40 is similar to the small-step rule for **let** expressions (Rule 3), except that the final value of the state is irrelevant. Rule 41 is defined similarly.

Rule 42 for renamings is straightforward, where $\ell s\Theta$ is the renaming Θ applied via (16) to each label in ℓs. Rule 43 states that a procedure call transitions to **nil** after evaluating the by-value parameters, retrieving the definition of procedure p from the context, and then executing the instantiated procedure body.

Most of the rules in Fig. 10 may be formulated in the big-step style similarly. However, as noted by Mosses [26], properly defining the semantics of an abrupt return from function calls is problematic in the big-step style. This is because in the general case we need to reason about partial (abruptly terminated) traces. It is, however, possible to handle the case where function bodies are purely sequential (do not contain concurrency), or are known to terminate, by extending Rule 35 for sequential composition to terminate abruptly if c_1 terminates abruptly, and adapting Rule 27 for the big-step style[2]. If a function body contains concurrency and a non-terminating loop, however, the rules we have presented do not provide the expected semantics, and it is not straightforward to handle this case. Following the reasoning of Mosses, we do not pursue this objective any further, as the small-step style is more appropriate in the presence of concurrency and abrupt termination.

On the relationship between the small-step and big-step rules

Having now defined both a small-step and big-step semantics it is worth discussing their relationship. The main point of difference is that the big-step semantics cannot be used to describe non-terminating executions, whereas the small-step semantics can. This can be a limitation in describing the behaviour of reactive systems, and reasoning about loop termination and abrupt termination. However, for terminating executions, the relationship is straightforward to define. We have that, for all traces ℓs,

$$c \xrightarrow{\ell s}\!\!\!\!\twoheadrightarrow \mathbf{nil} \quad \Leftrightarrow \quad c \Longrightarrow \mathbf{nil}$$

That is, if the big-step semantics determines that c terminates according to the trace ℓs, then there is a series of steps in the small-step semantics in which c terminates, generating a trace which consists of ℓs with a finite (possibly 0) number of τ steps interleaved.

That this relationship does indeed hold for all of the standard commands is straightforward; indeed, the big-step rules are based directly on the small-step equivalents. For example, take a sequential composition $(c_1 \,;\, c_2)$, and compare Rules 5 and 35, inductively assuming the relationship for c_1 and c_2. The only point of difference is that Rule 5(b) inserts a τ step between the execution of c_1 and c_2, but, as stated above, internal steps may be ignored for the purpose of proving equivalence.

In earlier work [7] we have provided a big-step expression evaluation semantics (but small-step command execution) for the language CSP_σ, which is an extension of the process algebra CSP [12] to include local state. In that paper, the basic

[2] Note, however, that these changes would compromise the desirable property of modularity [26], since the semantics of previously defined command types must be redefined in the presence of a new command.

state-based commands and labels are Morgan-style *specification statements* [24] of the form $\vec{x}$: [R], where the *frame* $\vec{x}$ is the list of variables that may be updated according to the relation R. Specification statements allow multiple variables to be updated nondeterministically, and in a single atomic step. In CSP$_\sigma$ an assignment $x := x + 1$ is an abbreviation for the specification statement x: [$x' = x + 1$], where x' indicates the post-state of x. A nondeterministic assignment of either 0 or 1 to x may be written x: [$x' \in \{0, 1\}$]. As a label, it is a generalisation of an atomic trace. For example, in this paper the trace of the evaluation of an expression $x + y < 5$ may be $(x = v_1).(y = v_2)$ where v_1 and v_2 are the values for x and y in context, respectively. In [7] the label for this expression is the specification command [$x + y < 5$] (the expression does not alter the state and hence the frame is empty). The local states successively replace variables in the predicate with their value in context, i.e., at the top level the label is [$v_1 + v_2 < 5$], which is equivalent to [*true*] or [*false*] depending on the values v_1 and v_2. Notably, the specification statement label may be nondeterministic in the new values for the variables in the frame. For example, the command x: [$x' \in \{0, 1\}$] has exactly one transition, with an identical label. However, in this paper the rules generate a set of individually deterministic labels. For example, in the above case, the command would generate two deterministic traces, $x := 0$ and $x := 1$. While the approach of [7] has some advantages, it is less suited to the situation where function calls can appear in expressions, and hence in this paper we have adopted the trace-based approach.

8. Command equivalence

We now provide a sketch of how command equivalence may be defined using the semantics. Perhaps the most intuitive notion for equivalence of commands c and d, written $c \approx d$, is that their traces are 'equivalent', by which we mean that their traces are identical if internal steps are removed. For the big-step semantics, this notion is straightforward to define, since the rules do not admit internal steps (a finite sequence of internal steps in the small-step semantics corresponds to an empty trace in the big-step). Hence, we require that all traces of each command are identical.

$$c \approx d \mathrel{\widehat{=}} \left(\forall\, \ell s \bullet c \xrightarrow{\ell s}\!\!\!\!\rightarrow \mathbf{nil} \Leftrightarrow d \xrightarrow{\ell s}\!\!\!\!\rightarrow \mathbf{nil} \right)$$

For instance, recall (18).

$$c \approx (\mathbf{state}\ \{x \mapsto v\} \bullet c) \quad \text{provided } x \text{ nfi } c, c \text{ does not contain procedure calls}$$

Proving the equivalence is straightforward: by the assumption that x does not appear free in c, and structural induction on the language constructs excluding procedure call, any trace ℓs generated by c cannot contain accesses of x. By the big-step rule for local states, Rule 41, the trace generated by (**state** $\{x \mapsto v\} \bullet c$) is $\ell s[\{x \mapsto v\}]$ which by (30) and assumption is equal to ℓs. Furthermore, the premise of Rule 41, $\mathit{consistent}(\ell s, \{x \mapsto v\})$, holds. Therefore the traces generated by each command are the same.

However, if we admit the atomic command, we require a more careful definition of program equivalence. In particular, we require

$$(\mathbf{atomic}\ x := 0\ ;\ y := 1) \approx (\mathbf{atomic}\ y := 1\ ;\ x := 0)$$

This is not immediate since their traces are different. However, an appropriate definition of equivalence in this case may be based on a notion of trace equivalence, with respect to any state, e.g., using Definition 9,

$$\ell s_1 \approx \ell s_2 \mathrel{\widehat{=}} (\forall\, \sigma : \mathit{State} \bullet \sigma[\ell s_1] = \sigma[\ell s_2]).$$

Since the order of the assignments to x and y does not affect their final values, the traces generated by the two atomic commands have the same effect on any state. An alternative approach to handling atomic traces is to use the more expressive label type in [7] as discussed in Section 7.

To define program equivalence using the small-step semantics requires dealing with the possibility of non-termination, and mapping internal steps of one command to zero or more internal steps of the other. In this case Milner's definition of *weak bisimilarity* [22], and strategies for proving weak bisimilarity, are applicable. A specialised scheme for proving program equivalence may incorporate state-specific reasoning [29] in addition to the pure trace-based reasoning of Milner.

9. On the use of locations

Plotkin [33, Section 3.4] identified four aspects of programming languages which decided him in favour of using locations in his semantics. In this section we provide a semantics that does not rely on locations for two of these aspects (static binding in Section 9.1 and arrays in Section 9.2). The remaining two aspects mentioned by Plotkin were by-reference parameters, which were dealt with in Section 4, and reference types (pointers). To describe reference types requires some notion of a global heap, and Plotkin naturally uses locations for this. An alternative is to explicitly model the heap as a global variable, which is amenable to our presentation. A semantics for Plotkin-style locations, including variable declarations and static binding, is given in Appendix A.

9.1. Static and dynamic binding

The effect of a procedure on its free variables, that is, variables that are neither declared locally nor in the parameter list, is determined by the *binding* strategy the language employs. There are two main strategies, *dynamic* and *static*. With *dynamic binding*, procedure and variable names refer to the most recently instantiated version at run-time. With *static binding*, procedure and variable names refer to the closest enclosing definition in the code (for more detail, see, for instance, Sebesta [40]).

For example, consider the following program in which procedure q locally declares a variable x, which is also declared globally.

$$\begin{array}{l} \texttt{var}\ x\ ;\ x := 0\ ; \\ \texttt{proc}\ p() \mathrel{\widehat=} (x := 1)\ ; \\ \texttt{proc}\ q() \mathrel{\widehat=} (\texttt{var}\ x\ ;\ x := 2\ ;\ p())\ ; \\ q() \end{array} \tag{33}$$

The assignment $x := 0$ refers to the globally declared x, while the assignment $x := 2$ in the body of q refers to the version of x declared locally in q. The interesting question is to which version of x does the assignment $x := 1$ in p refer. Following a static binding strategy, it refers to the globally declared x, and hence after calling q (which calls p), the value of the global x will be 1. That is, the version of x to which p refers depends on the context in which p is declared. Following a dynamic binding strategy, the free reference to x in p will be captured by the local declaration in q, and hence the final value of the global x will remain 0. That is, the version of x to which p refers depends on the context in which p is called.

Dynamic binding is a powerful feature but can cause subtle and hard-to-diagnose problems, because the references may not be determined by inspection. As a consequence, well-formedness is undecidable in general since it is not always possible to statically determine the types of actual parameters. In contrast, with static binding it is much easier to understand the code by inspection, and indeed, almost all practical languages have a static binding strategy (a dynamic strategy was used in Lisp [21] before later changing to static for these reasons).

The rules for procedures we have given account for the semantics of dynamic binding. However, as outlined above, the ambiguity in the references is due to multiple declarations using the same identifier (in the above example (33), the identifier x is used for two different variable declarations). Such name clashes must be resolved, and this is straightforward to achieve *prior* to execution, that is, statically. For instance, a simple strategy of renaming a variable x declared locally to a procedure q to $q.x$ disambiguates the local x from the global occurrence.

$$\begin{array}{l} \texttt{var}\ x\ ;\ x := 0\ ; \\ (\texttt{proc}\ p \bullet () \mathrel{\widehat=} x := 1)\ ; \\ (\texttt{proc}\ q \bullet () \mathrel{\widehat=} \texttt{var}\ q.x\ ;\ q.x := 2\ ;\ p())\ ; \\ q() \end{array} \tag{34}$$

Here it is clear that the global x and the local $q.x$ are distinct, and that the reference to x in p refers to the global variable. The renaming of variables to avoid clashes is a straightforward syntactic modification using a set of fresh variable names. In the above case the renaming was simplified because the inner scope had an obvious name (the name of the declaring procedure), however providing a unique name for anonymous scopes (such as the scope of iterator variables in **while** loops) is also straightforward. The relationship between the two programs (33) and (34) is obvious, and (33) may be used as the concrete syntax for (34). Indeed, a common approach of compilers is to give each declaration block a unique name, and each identifier in that block a unique offset. The combination is the unique name for each variable, regardless of the identifier used in the concrete code.

We believe it is both simpler and more intuitive to resolve the ambiguity with variable names statically (prior to execution) rather than dynamically (during execution). However, a dynamic approach is certainly valid, and several options for implementing the renaming dynamically exist. Indeed, Plotkin presents such a solution: ambiguity is eliminated by dynamically renaming variable identifiers to locations, which are for these purposes a set of fresh "variable" names. The global store is a mapping from locations to values. For completeness, in Appendix A we give an encoding of Plotkin-style locations and the dynamic semantics of static binding.

We note that the renaming strategy outlined above does not resolve a related, but separate, issue to do with identifiers. During execution, it is possible for there to be multiple instances of the same declaration, and hence, dynamically, the same identifier may appear multiple times. There are several ways this may occur, for instance, if two concurrent processes call the same procedure at the same time, then there will be two copies of the formal parameters and local variables, one copy in each process. Another way for this to occur is through recursion, since recursive calls are nested, and hence create multiple nested instances of the formal parameters and local variables. These clashes must be dealt with dynamically. In Section 3.6.3 we demonstrated that local states in parallel threads do not cause interference even if they use the same name. In the recursive case, recall (25) in Section 4.5.3 in which the execution of the recursive procedure *sum* has generated three nested instances of the local variable a (a parameter). The value for each a is kept locally, and because each nested instance of *sum* refers to its

Assume $cv, i, \mathsf{n} \in Val \quad lve \in LValExpr$

Rule 44 (**Array index evaluation**).

$$\frac{e_1 \overset{\ell s_1}{\longrightarrow\!\!\!\!\!\rightarrow} cv \quad e_2 \overset{\ell s_2}{\longrightarrow\!\!\!\!\!\rightarrow} i \quad i \in \mathrm{dom}(cv)}{e_1[e_2] \overset{\ell s_1.\ell s_2}{\longrightarrow\!\!\!\!\!\rightarrow} cv(i)}$$

Rule 45 (**Record field evaluation**).

$$\frac{e \overset{\ell s}{\longrightarrow\!\!\!\!\!\rightarrow} cv \quad \mathsf{n} \in \mathrm{dom}(cv)}{e.\mathsf{n} \overset{\ell s}{\longrightarrow\!\!\!\!\!\rightarrow} cv(\mathsf{n})}$$

Rule 46 (**L-Value evaluation**).

$$(a) \quad x \longrightarrow_{\mathsf{lv}} x(\epsilon) \qquad (b) \ \frac{lve \overset{\ell s_1}{\longrightarrow}_{\mathsf{lv}} x(\vec{v}) \quad e \overset{\ell s_2}{\longrightarrow\!\!\!\!\!\rightarrow} v}{lve[e] \overset{\ell s_1.\ell s_2}{\longrightarrow}_{\mathsf{lv}} x(\vec{v}.v)} \qquad (c) \ \frac{lve \overset{\ell s}{\longrightarrow}_{\mathsf{lv}} x(\vec{v})}{lve.\mathsf{n} \overset{\ell s}{\longrightarrow}_{\mathsf{lv}} x(\vec{v}.\mathsf{n})}$$

Fig. 13. Rules for arrays.

own instance of a, modifications of a by the code may be calculated on the closest enclosing local state. Hence the ambiguity due to multiple instances of the same identifier occurring dynamically is resolved by the scope of each (nested) identifier.

It is worth noting that, due to the potential for there to exist multiple instances of a variable during execution even if all declarations use unique identifiers, a scheme in which the values of variables are maintained globally requires some renaming to disambiguate those instances. In Plotkin-style semantics, the use of locations also fulfills this role.

9.2. Arrays and records

In this section we consider the extensions necessary to allow array and record types, and in particular, passing array indexing and field access expressions as reference parameters. In an implementation setting, individual entries in an array or record can be accessed directly through their mapping to locations. In our framework we provide a semantics for interpreting updates to array indexes and record field accesses as overriding the value of a function at an index.

9.2.1. Basics and syntax

We refer to arrays and records as special elements of *Val* called *compound values* (they are mathematical functions, but we use the term compound value to distinguish them from command-level functions and function denotations). A compound value representing an array of length i has a domain $0..(i-1)$, and a compound value representing a record has a domain consisting of some set of constants in *Val* (the field names). The range of a compound value may be basic values or compound values, and hence may be a recursive structure.

The syntax of expressions is extended to allow array indexing and record field access.

$$e ::= \ldots \mid e_1[e_2] \mid e.\mathsf{n}$$

Here e_2 is an expression assumed to be integer-valued, and n is a field name, which are elements of *Val*[3]. This syntax allows arrays of arrays, arrays of records, etc. For instance, an array A of records with field name n can be accessed through expressions such as $A[0].\mathsf{n}$. Rules 44 and 45 in Fig. 13 give the big-step evaluation rules for array index and field access expressions, respectively. We have chosen the big-step style for compactness, and because we will not be concerned with termination issues in this section; the corresponding small-step rules are straightforward to derive.

9.2.2. L-values

The type *LValExpr*, of L-value expressions, represents expressions that may appear on the left-hand side of assignments or as actual by-reference parameters to procedure calls. So far we have allowed variables as the only kind of L-value expression, but we now extend L-value expressions to include indexes into arrays and field access of records. The syntax of an L-value expression $lve \in LValExpr$ is given by the following production.

$$lve ::= x \mid lve[e] \mid lve.\mathsf{n}$$

[3] We are assuming that field names may not be constructed from general expressions, although this is not a restriction that simplifies the operational semantics particularly. However, the restriction does simplify the static semantics of expressions, in particular type checking.

Hence, $A[y].\mathsf{n}$ is an *LValExpr*, but $(x + y).\mathsf{n}$ is not. The syntax of commands is extended to allow *LValExpr*s to appear on the left-hand side of assignments, and in the actual by-reference parameters of procedure calls.

$$c ::= \ldots \mid lve := e \mid \ldots \mid p(\vec{lve}, \vec{e}) \mid \ldots$$

Our intention is that an update of a one-dimensional array, $A[0] := v$, has the semantic effect of the update $A := A[0 \mapsto v]$. That is, the compound value stored in A is updated so that its first element becomes the value v, following the syntax of updating a state. For assignment statements, it would be possible to use the $A[0] := v$ notation as a shorthand for $A := A[0 \mapsto v]$, however, this does not extend easily to allowing array index expressions as by-reference parameters.

We introduce a new syntactic class *LVal*, which corresponds to *LValExpr*s where all but the main variable have been evaluated. We denote them by the pairing of a variable with a list of values that are the indexes into the structure of the variable.

$$LVal \mathrel{\widehat=} Ident \times \mathbf{list}\ Val$$

We write an *LVal* with identifier x and index list $\vec{v}$ as $x(\vec{v})$. An *LVal* $x(\epsilon)$ represents a variable with a basic value, that is, there is no indexing into its structure. An *LVal* $x(\langle i \rangle)$ represents a one-dimensional array being accessed at its ith element. An *LVal* $x(\langle i, \mathsf{n} \rangle)$ represents accessing fieldname n of the ith element in an array of records, and so on.

An *LValExpr* may be evaluated to an *LVal* via the relation $\longrightarrow_{\mathsf{lv}}$, which is the least relation that satisfies Rule 46. The rule is given in big-step style. A variable x is evaluated to an *LVal* with an empty index list, while array indexes and fieldnames are appended in left-to-right order to the index list. For example, the expression $x[0]$ is evaluated to the *LVal* $x(\langle 0 \rangle)$, and the expression $x[0].\mathsf{n}$ is evaluated to the *LVal* $x(\langle 0, \mathsf{n} \rangle)$. Lists of *LVal*s may be evaluated analogously to Rule 20 for lists of expressions.

9.2.3. Labels and states

We generalise labels so that *LVal*s may appear in place of variables.

$$\ell ::= \tau \mid x(\vec{v}) = v \mid x(\vec{v}) := v$$

A variable x in the original labels is represented by $x(\epsilon)$ in the new labels. The updated label types necessitate corresponding generalisations of the definitions for the local state functions in Definition 9. We first introduce notation for calculating the value of a compound value at a given index, $eval(cv, \vec{v})$, and updating a compound value at a given index, $cv[\vec{v} := v]$.

$$\begin{aligned} eval(cv, \epsilon) &= cv & cv[\epsilon := v] &= v \\ eval(cv, i.\vec{v}) &= eval(cv(i), \vec{v}) & cv[i.\vec{v} := v] &= cv[i \mapsto (cv(i)[\vec{v} := v])] \end{aligned}$$

We note that a compound value may be thought of as either a simple value or a list of compound values (i.e., an array of values, an array of arrays, etc.). A compound value, cv, given a list of indexes, $\vec{v}$, is evaluated by recursively traversing the structure of cv in tandem with $\vec{v}$. Similarly, an update of a compound value is calculated by updating the first index by the updated version of the compound value at that index. If the list of indexes $\vec{v}$ is empty, then cv is a simple value (or a compound value being updated in its entirety), and hence cv is updated in it entirety to v. As an example, consider an update $cv[\langle 1, \mathsf{n} \rangle := v]$, where cv is a one-dimensional array of records (this update corresponds to an assignment '$x[1].\mathsf{n} := v$' in concrete syntax, where x is a variable with value cv). The new value is constructed below:

$$\begin{aligned} & cv[\langle 1, \mathsf{n} \rangle := v] \\ = \ & cv[1 \mapsto (cv(1)[\langle \mathsf{n} \rangle := v])] \\ = \ & cv[1 \mapsto (cv(1)[\mathsf{n} \mapsto (cv(1)(\mathsf{n})[\epsilon := v])])] \\ = \ & cv[1 \mapsto (cv(1)[\mathsf{n} \mapsto v])] \end{aligned}$$

Hence, the new value is cv with its first record replaced by the original first record ($cv(1)$) updated so that field $\mathsf{n} = v$.

We now give a version of Definition 9 for the new label type that uses *LVal*s. The definitions remain essentially the same, except that calculating the new value in the local state and ensuring consistency requires slightly more work for compound values. In all cases, if the label refers to a non-compound valued variable x, i.e., if the index list $\vec{v}$ is empty, the definitions collapse to those in Definition 9

$$\ell[\sigma] = \begin{cases} \tau & \text{if } \ell \text{ is } x(\vec{v}) = v \text{ and } x \in \mathrm{dom}(\sigma) \\ \tau & \text{if } \ell \text{ is } x(\vec{v}) := v \text{ and } x \in \mathrm{dom}(\sigma) \\ \ell & \text{otherwise} \end{cases}$$

Rule 47 (**State – test**).

$$\frac{c \xrightarrow{x(\vec{v})=v} c' \quad x \in \mathrm{dom}(\sigma) \quad eval(\sigma(x), \vec{v}) = v}{(\mathbf{state}\ \sigma \bullet c) \longrightarrow (\mathbf{state}\ \sigma \bullet c')}$$

Rule 48 (**State – assignment**).

$$\frac{c \xrightarrow{x(\vec{v}) := v} c' \quad x \in \mathrm{dom}(\sigma)}{(\mathbf{state}\ \sigma \bullet c) \longrightarrow (\mathbf{state}\ \sigma[x \mapsto \sigma(x)[\vec{v} := v]] \bullet c')}$$

Fig. 14. Derived small-step rules for L-values and local state.

Assume $\vec{lve} \in \mathbf{list}\ LValExpr \quad \vec{lv} \in \mathbf{list}\ LVal$

Rule 49 (**Assignment**).

$$\frac{lve \xrightarrow{\ell s_1}_{\mathsf{lv}} x(\vec{v}) \quad e \xrightarrow{\ell s_2}\!\!\!\!\rightarrow v}{lve := e \xrightarrow{\ell s_1.\ell s_2.(x(\vec{v}) := v)}\!\!\!\!\rightarrow \mathbf{nil}}$$

Rule 50 (**Procedure call**).

$$\frac{\vec{lve} \xrightarrow{\ell s_1}_{\mathsf{lv}} \vec{lv} \quad \vec{e} \xrightarrow{\ell s_2}\!\!\!\!\rightarrow \vec{v} \quad \mathsf{p}_{\mathsf{den}}(\vec{lv})(\vec{v}) \xrightarrow{\ell s_3}\!\!\!\!\rightarrow \mathbf{nil}}{p(\vec{lve}, \vec{e}) \xrightarrow{\ell s_1.\ell s_2.(p=\mathsf{p}_{\mathsf{den}}).\ell s_3}\!\!\!\!\rightarrow \mathbf{nil}}$$

Fig. 15. Big-step rules for assignment and procedure call using *LVals*.

$$\sigma[\ell] = \begin{cases} \sigma[x \mapsto \sigma(x)[\vec{v} := v]] & \text{if } \ell \text{ is } x(\vec{v}) := v \text{ and } x \in \mathrm{dom}(\sigma) \\ \sigma & \text{otherwise} \end{cases}$$

$$consistent(\ell, \sigma) \Leftrightarrow \begin{cases} eval(\sigma(x), \vec{v}) = v & \text{if } \ell \text{ is } x(\vec{v}) = v \text{ and } x \in \mathrm{dom}(\sigma) \\ true & \text{otherwise} \end{cases}$$

Using these definitions, the small-step and big-step general rules for local states, Rules 14 and 41, apply for *LVals*. To make the relationship between *LVals* and local states clearer, in Fig. 14 we give Rules 47 and 48, which are derived small-step rules for checking the value of an *LVal* and updating an *LVal*, respectively. They structurally match with Rule 11(a) and Rule 12(a).

We must also redefine (16) for the new label types, to define the effect of a renaming on the array-based labels (via Rules 18 and 42).

$$\ell\Theta = \begin{cases} y(\vec{v}_1.\vec{v}_2) = v & \text{if } \ell \text{ is } x(\vec{v}_2) = v \text{ and } x \in \mathrm{dom}(\Theta) \text{ and } \Theta(x) = y(\vec{v}_1) \\ y(\vec{v}_1.\vec{v}_2) := v & \text{if } \ell \text{ is } x(\vec{v}_2) := v \text{ and } x \in \mathrm{dom}(\Theta) \text{ and } \Theta(x) = y(\vec{v}_1) \\ \ell & \text{otherwise} \end{cases} \tag{35}$$

For instance, consider a command $(\mathbf{rn}\ \{x \mapsto y(1)\} \bullet x.\mathsf{n} := 0)$, which assigns the value 0 to field n of x, where x is an alias for the first element in array y. The assignment generates the label $x(\mathsf{n}) := 0$, and in this instance $\Theta(x) = y(1)$. Hence the renamed label is $y(1.\mathsf{n}) := 0$, which, as expected, updates field n of element 1 of the array y. For renaming of non-indexed variables, where both $\vec{v}_1$ and $\vec{v}_2$ are empty, equation (35) collapses to (16).

9.2.4. Commands

The commands and expressions that are affected by the introduction of *LVals* are the assignment command and procedure and function calls (see Fig. 15). Rule 49 subsumes Rule 34, adding an extra trace ℓs_1 to evaluate the *LValExpr* on the left-hand side of the assignment, and Rule 50 is a big-step version of Rule 19, with the additional requirement to evaluate the by-reference parameters to *LVals*. The rule for function calls is similar, although note that the big-step rule style for function calls can be problematic in the presence of concurrency, as discussed in Section 7.

The instantiation of the procedure or function occurs in the usual way (Rule 50), except that now the procedure denotations include renamings from *Ident* to *LVal*. Therefore, context-dependent behaviour involving the formal by-reference parameters (which must always be variables) such as $x = v$ and $x := v$, will be renamed to $x(\vec{v}) = v$ and $x(\vec{v}) := v$.

As an example of using *LVal*s as actual by-reference parameters, consider the following code that uses the procedure *square* (14). We let the compound value $\{0 \mapsto v_0, 1 \mapsto v_1, \ldots\}$, representing an array, be abbreviated by $\langle\!\langle v_0, v_1, .. \rangle\!\rangle$.

$$(\mathbf{state}\ \{A \mapsto \langle\!\langle -1, 3 \rangle\!\rangle\} \bullet (\mathbf{procdecl}\ \{square \mapsto \mathsf{sq}_{\mathsf{den}}\} \bullet square(A[0], A[1])))$$

The result of the call is to update $A(0)$ to 9, i.e., to update A to $\langle\!\langle 9, 3 \rangle\!\rangle$.

The expression $A[1]$ is evaluated to 3 using Rule 44. Recalling that the range of renamings is now *LVal*, the expansion of the procedure call in context gives:

$$\mathbf{rn}\ \{z \mapsto A(0)\} \bullet (\mathbf{state}\ \{x \mapsto 3\} \bullet z := x^2)$$

As shown in (21), the context-dependent behaviour of this command becomes

$$A(0) := 9$$

Through Rule 48 and the calculation below, this has the desired effect of changing the value of A in the local state from $\langle\!\langle -1, 3 \rangle\!\rangle$ to $\langle\!\langle 9, 3 \rangle\!\rangle$.

$$\begin{aligned} & \langle\!\langle -1, 3 \rangle\!\rangle[\langle 0 \rangle := 9] \\ = \ & \langle\!\langle -1, 3 \rangle\!\rangle[0 \mapsto -1[\epsilon \mapsto 9]] \\ = \ & \langle\!\langle -1, 3 \rangle\!\rangle[0 \mapsto 9] \\ = \ & \langle\!\langle 9, 3 \rangle\!\rangle \end{aligned}$$

The syntax of the aliasing construct of Plotkin [33, Section 3.4] may be extended for arrays by allowing aliasing of *LVal* expressions, '$x == lve\ ;\ c$'. Conceptually, such a command in our language is executed by evaluating *lve* via Rule 46 to some *LVal* lv, and elaborating the construct into a renaming ($\mathbf{rn}\ \{x \mapsto lv\} \bullet c$). A big-step version of this rule appears on the left below; on the right we also give a small-step version of the rule, where for descriptive purposes we assume the *LVal* expression may be evaluated atomically (the non-atomic version is straightforward).

$$\frac{lve \xrightarrow{\ell s_1}_{\mathsf{lv}} lv \quad \Theta = \{x \mapsto lv\} \quad c \xrightarrow{\ell s_2}\!\!\!\!\!\rightarrow \mathbf{nil}}{(x == lve\ ;\ c) \xrightarrow{\ell s_1.(\ell s_2 \Theta)}\!\!\!\!\!\rightarrow \mathbf{nil}} \qquad \frac{lve \xrightarrow{\ell s}_{\mathsf{lv}} lv}{(x == lve\ ;\ c) \xrightarrow{\ell s} (\mathbf{rn}\ \{x \mapsto lv\} \bullet c)}$$

10. Related work

Plotkin's *syntax-directed* operational semantics was a great improvement over earlier lower-level stack-machine descriptions. A comprehensive example of language definition using a Plotkin-style big-step operational semantics is given by Milner et al. for Standard ML [23]. Surveys on current trends in operational semantics research are given by Mousavi et al. [30] and Aceto et al. [2]. The historical development of operational semantics is discussed by Plotkin [34] and by Jones [15]. In this section we compare our framework with that of Plotkin [33], Mosses [26], and other related work.

10.1. Plotkin-style SOS

In the full rules (handling procedures) for Plotkin-style SOS [33], each transition rule is defined on a configuration that contains a command, a store (mapping from locations to values), and an environment (mapping from identifiers to locations). In contrast, our rules are defined on commands only, with the addition of context-dependent information appearing on the transition label. This information includes conditions on the state and updates of the state (the primitive actions of a program).

If we ignore for the moment the environment in Plotkin-style rules, and hence assume that the store maps identifiers directly to their values, we may relate the two semantics as follows. In Plotkin-style semantics a command c is executed in a configuration, $\langle c, \sigma \rangle$, where σ is a state containing a mapping for at least every variable used in c. In the small-step semantics, the successive configurations that follow from $\langle c, \sigma \rangle$ match with the successive local states and commands that result from executing ($\mathbf{state}\ \sigma \bullet c$) in our semantics. To make this clearer, below we show the Plotkin-style transitions and their counterparts in our style. Because all of c's execution involves variables 'local' to σ, each step in our semantics is internal.

$$\begin{array}{ccccccc} \langle c, \sigma \rangle & \longrightarrow & \langle c_1, \sigma_1 \rangle & \longrightarrow & \ldots & \longrightarrow & \langle c', \sigma' \rangle \\ (\mathbf{state}\ \sigma \bullet c) & \xrightarrow{\tau} & (\mathbf{state}\ \sigma_1 \bullet c_1) & \xrightarrow{\tau} & \ldots & \xrightarrow{\tau} & (\mathbf{state}\ \sigma' \bullet c') \end{array}$$

More succinctly, using $\Rightarrow$ to bundle a sequence of small-steps, we have the following relationship between Plotkin-style SOS and an execution with our rules.

$$\langle c, \sigma \rangle \Longrightarrow \langle c', \sigma' \rangle \quad \Leftrightarrow \quad (\mathbf{state}\ \sigma \bullet c) \Longrightarrow (\mathbf{state}\ \sigma' \bullet c')$$

We may expand the right-hand side as follows, by considering the effect of repeated applications of the small-step rule for local state, Rule 14 (cf. Rules 29 and 41). Hence, $\langle c, \sigma \rangle \Longrightarrow \langle c', \sigma' \rangle$ if and only if

$$(\exists\, \ell s \bullet c \xRightarrow{\ell s} c' \land \mathit{consistent}(\ell s, \sigma) \land \sigma[\ell s] = \sigma')$$

This states that any execution of $\langle c, \sigma \rangle$ in Plotkin's rules is mirrored by an execution of c in our rules that generates some trace ℓs, which is consistent with σ, and the updates in ℓs when applied to σ yield σ'.

The above definitions relate the styles in the case where environments (and hence locations) are not used. Plotkin introduces locations to cope with several aspects of programming (see Section 9). We give a semantics for Plotkin-style locations in Appendix A. However, within the body of the paper we have provided a semantics that does not use locations, due to the interaction of the transition labels ($x = v$ and $x := v$) with the local state and renaming command types. Below we consider reformulating these constructs in a Plotkin-style semantics without locations.

Our local state command simplifies some reasoning since local modifications have a localised effect. This is particularly useful in the situations where multiple instances of the same declaration occur, for instance, in concurrent threads that each execute a block of code that declares a variable of the same name, and when nested instances occur through recursion (see Section 9.1 for more detail). A rule for local state is given in Plotkin-style notation by Reynolds [35, Section 6.2] (attributed to Eugene Fink). For comparison, we give a version of Reynold's rule in our syntax below, where γ is the local state and σ is the global state, and a Plotkin-style restriction, '$\sigma \upharpoonright \vec{x}$', is the function σ restricted in its domain to just those variables in $\vec{x}$.

$$\frac{\langle c, \sigma \oplus \gamma \rangle \longrightarrow \langle c', \sigma' \rangle}{\langle (\mathbf{state}\ \gamma \bullet c), \sigma \rangle \longrightarrow \langle (\mathbf{state}\ (\sigma' \upharpoonright \mathrm{dom}(\gamma)) \bullet c'), \sigma' \oplus (\sigma \upharpoonright \mathrm{dom}(\gamma)) \rangle}$$

We assume that the domain of σ contains all required identifiers, including those declared locally in γ, and hence $\mathrm{dom}(\sigma) = \mathrm{dom}(\sigma \oplus \gamma) = \mathrm{dom}(\sigma')$. Note that in the final configuration the original values for global variables in the domain of the local state are restored to their initial values in σ.

The dynamic renaming command ($\mathbf{rn}\ \Theta \bullet c$) (and, therefore, Plotkin's aliasing construct $x == y\ ;\ c$), is more problematic to define in Plotkin-style semantics without using locations. One is tempted to give a straightforward unfolding of c in which all variables captured by Θ are renamed.

$$\langle (\mathbf{rn}\ \Theta \bullet c), \sigma \rangle \longrightarrow \langle c[\Theta], \sigma \rangle$$

The command $c[\Theta]$ specifies textual substitution of identifiers x in the domain of Θ with $\Theta(x)$. However this runs into the common problem of variable capture, e.g., if c redeclares one of the variables in $\mathrm{dom}(\Theta)$. To handle this situation requires a further renaming of c to use unique names for its local variable and procedure declarations. When fresh names are introduced dynamically, it appears to be more convenient to use locations, as in Plotkin-style semantics, where the local environments correspond to our local renamings. This also has the advantage that the original text of the command remains intact. The disadvantage of using locations is the complexity introduced by relating variables to their values indirectly through the environment.

The encapsulation of the effect of the by-value parameters (as a local state) and the by-reference parameters (as a (local) renaming) provides a separation of concerns for defining the semantics of procedure calls. For the semantics of both local states and dynamic renaming, summarising the effect of the transition in the label as either a condition on the state (guard $x = v$) or modification of the state (update $x := v$) is simpler than using a pair of global states (pre and post) in the rule configurations. These labels represent the level of atomicity that we have chosen for the semantics of the language, although other choices could have been made; a more general label type that subsumes both guard and update is explored in the context of the process algebra CSP [7], and is discussed in more detail in Section 7.

10.2. Modular Structural Operational Semantics (MSOS)

Mosses' MSOS [26] (see also [25,27]) is a powerful technique for simplifying the expression of operational semantics transition rules, and in particular for ensuring that rules may be reused without change as the complexity of the language increases. Mosses calls this property *modularity*; in the context of Plotkin's SOS, it means that the simple rules for each language construct, e.g., sequential composition, do not need to be rewritten as information, such as the environment, is added to the semantics.

Modularity in MSOS is achieved through using a rich type for labels. Labels may include information that normally appears in configurations, such as the store and the environment, as well as other information, such as exceptions, or, in a process-algebraic setting, synchronisation actions. Only those parts of the label that are accessed or modified need be

specified in a rule. Hence, in the case of sequential composition, for example, the rule does not depend on the context, and the label is left unconstrained. This means that although the complexity of the label increases as the language is extended, the sequential composition rule remains valid. The trace of a command is formed by choosing a sequence of transitions for which successive labels are consistent (i.e., the post-state of a label matches the pre-state of the next).

MSOS has been used to modularise and simplify Plotkin-style rules, in particular, reducing the size of the configurations from three elements (command, state, and environment) to one element (command). The MSOS framework may be used to express our rules, since its generality applies to any SOS format. However, since our configurations are singleton, as it stands there would be little advantage in rewriting the rules in MSOS style. However, in general, a label in MSOS contains a slot for each possible operation or context information, while our labels refer to only one, and hence combining operations that require multiple labels may be better facilitated by using the MSOS framework. A recent development, Implicitly Modular SOS (*I-MSOS*) [28], admits a style of SOS in which label information is implicitly propagated; employing this style in our framework would further improve conciseness of the rules by eliminating the need to write the label ℓ in rules such as Rule 5(a).

10.3. Other related work

The use of labelled transitions in operational semantics is widespread in process calculi (e.g., [2,6,12,22]). The labels in such languages contain the "observable behaviour" of processes, which are typically process interactions. In this paper we have applied this approach to a state-based imperative language, where state tests and assignments are the obvious candidates to appear in labels. We have also taken the language further towards the process algebra world by allowing partial states to "hide" local computation, in an analogous manner to the way synchronisation events may be hidden in process calculi.

Baeten and Bergstra [4,5] describe a process algebra with a local state operator which is tested and updated through propositional signals and abstract actions. The state is anonymous, but may emit a propositional signal that can be used to query the current state (such as *empty* or *nonempty*, for the state of a stack), and can be used by subprocesses as guards. The state is updated by defining the effect of abstract actions on the state, e.g., if the state is a stack, and the subprocess transitions with action $push(a)$, the effect on the state may defined to add a to the top of the stack. Our local state command is similar to Baeten and Bergstra's, however, due to the anonymity of the state, and communication occurring through propositional variables rather than directly on the state, our approach appears to be simpler for defining the semantics of standard imperative code.

Owens [31] gives an operational semantics for OCaml light that uses similar basic label types to ours to handle testing and updating the global store. The use of these labels means that the store does not need to appear in the majority of the rules, resulting in a more concise and modular semantics. Our (independent) work takes this idea further by giving a semantics for local states, and also using labels for retrieving procedure definitions returning from functions, and defining the meaning of by-reference parameters in procedure calls.

An operational semantics for a hybrid process-algebraic formalism is provided by van Beek et al. [42]. This is a complex language which allows continuous real-time variables, delay predicates and algebraic variables, along with familiar process algebraic operators for concurrency and choice. The semantics is defined on configurations which include multiple valuations for different types of variables, and labels which admit both synchronisation actions and pre- and post- valuations for variables. The semantics also allows propositional signals as in [5]. Interesting future work is to test whether the SOS style we present can handle all of the different operation types required by hybrid processes, and hence help to make the transition rules more concise and modular.

In previous work [7] we used our approach for introducing state tests and assignments to define an extension of CSP [12], called CSP_σ. That paper includes a more general state modification construct, the *specification command* [24], that can nondeterministically and atomically update several variables at a time. The transition labels are pairs which include both specification commands and synchronisation events, and hence allow operations on the state to be synchronised using the existing CSP operators. The specification command labels generalise this paper's $x = v$ and $x := v$ labels (see also the discussion at the end of Section 7). The language CSP_σ has been used to formalise a notation for capturing requirements [8].

The *Tile Model* of Gadducci and Montanari [9] introduces a format for writing transition rules that is extremely compact. Using their format, configurations (*Cmd* in our case, and $Cmd \times State$ for Plotkin-style SOS) are replaced by operators of the language, and the labels on transitions are relations that summarise the transitions of subprocesses. The disadvantage of this format is that the rules bear less resemblance to the syntax of imperative programs.

11. Conclusions

In the first part of this paper we present a small-step operational semantics for an imperative programming language with concurrency, procedure and function calls, and recursion. In the second part we consider some further aspects of programming language semantics. The semantic approach is novel in that transitions are labelled with the atomic action that was performed on the variables in the state, e.g., one label specifies the test of a variable, $x = v$, and another specifies the update of a variable, $x := v$. In this view, the meaning of a program is given by the set of traces it may produce, where traces are lists of labels.

Exposing the basic operations in the labels leads to a relatively straightforward definition of local states, which helps to keep the effect of local variables local. It also allows a straightforward definition of the semantics of by-reference parameters to procedure calls. As a consequence, Plotkin-style locations were not required to handle by-reference parameters, and we provided other mechanisms for giving the semantics of programming language features that traditionally use locations. Since states are maintained locally in the abstract syntax, and locations are not used, the configurations of the rules are singleton (containing only a command), in contrast to the triples used in Plotkin-style SOS (containing a command, environment and store). This makes it easier to maintain modularity of the rules [26], and potentially to apply general theorems about SOS rule formats [11,14,18,41] for automatically deriving properties of the language (such as distributivity of particular operators).

Since, we argue, the semantics of procedure calls is simpler in our framework, a proof of soundness of Hoare logic with procedures may be more straightforward [17,43]. The development of the notion of program equivalence may also be interesting to explore further: being trace-based, the semantics lends itself to the bisimulation techniques of Milner [22], with the inclusion of state-specific reasoning [29]. The local variable scoping we employ and the lack of interference through a global store, along with the explicit label type, simplified the operational semantics for a standard imperative language with concurrency and procedures; current research into programming languages uses more complex concepts which provide a more thorough test of a semantic framework, for instance, separation logic for pointer-based programs [13,36,37], memory models for concurrent programming [1], and language interoperability [20].

Acknowledgements

We would like to thank Kirsten Winter and Brijesh Dongol for feedback on earlier drafts of this paper, and three anonymous referees for their constructive comments which have improved the paper and our own understanding of programming language semantics. This work is supported, in part, by the Australian Research Council (ARC) Linkage Grant LP0989363.

Appendix A. Plotkin-style locations

In this section we further develop the comparison with Plotkin's work given in Section 10.1 by outlining the encoding of the Plotkin-style semantics for locations in our framework, including variable and procedure declarations and static binding.

A.1. Basics

We assume a new set *Loc* of locations, and define the following types:

$$Store : Loc \pfun Val \qquad Env : Ident \pfun (Loc \mid Val)$$

The type *Store* is similar to type *State* except that its domain is locations. The type *Env* is a mapping from identifiers to locations or values. The set of values includes *closures*, which in this context are procedure denotations that include the environment in which they are declared. Hence, an *Env* maps an identifier to a location if it represents a variable, or to a closure if it is a procedure identifier.

We extend the command syntax to allow a global *Store* and distributed environments, as in Plotkin [33]. We also introduce concrete syntax for declaring variables and their initial values, and a procedure.[4]

$$\begin{array}{l} \mathit{Assume} \quad \sigma \in \mathit{Store} \quad \rho \in \mathit{Env} \\ c ::= \ldots \mid (\mathbf{store}\ \sigma \bullet c) \mid (\mathbf{env}\ \rho \bullet c) \mid (\mathtt{vars}\ \vec{x} := \vec{v} \bullet c) \mid (\mathtt{proc}\ p(\vec{z}, \vec{x}) \mathrel{\widehat=} c_1 \bullet c_2) \\ \ell ::= \tau \mid x = v \mid x := v \mid l = v \mid l := v \mid \mathsf{newlocs}(\vec{l}, \vec{v}) \mid \mathit{env}(\rho) \end{array}$$

The labels are extended to include: operations on locations, $l = v$ and $l := v$; a label $\mathsf{newlocs}(\vec{l}, \vec{v})$ which is allowed by the **store** context if the elements of $\vec{l}$ have not already been used, and the store is updated to map the new locations to values in $\vec{v}$; and a label $\mathit{env}(\rho)$ where ρ is the local environment to a command.

The intention is that each top-level command, c, initially appears in the context of an empty store, $(\mathbf{store}\ \varnothing \bullet c)$, as with the initial configuration in a Plotkin-style execution. The store increases in size as variables declarations are processed.

In the rest of this section we provide rules for the new command types, which appear in Fig. A.1. Note that all rules presented so far stay the same, with one minor exception described in detail below: the local state rules become rules for the global store, which in addition requires a rule for adding new locations. We also provide rules for environments (which replace renamings), and elaboration of variable and procedure declarations. Rules for expression evaluation and command execution remain the same.

[4] The generalisation of these commands to allow variable initialisation to a list of expressions and declaring a list of procedures is straightforward.

Assume $l \in Loc \quad n \in \mathbb{N}$

Rule 51 (**Variable declaration**).

$$(\texttt{vars}\ \vec{x} := \vec{v} \bullet c) \xrightarrow{\text{newlocs}(\vec{l},\vec{v})} (\mathbf{env}\ (\vec{x} \mapsto \vec{l}) \bullet c)$$

Rule 52 (**Allocate**).

$$\frac{c \xrightarrow{\text{newlocs}(\vec{l},\vec{v})} c' \quad \vec{l} \cap \text{dom}(\sigma) = \varnothing}{(\mathbf{store}\ \sigma \bullet c) \xrightarrow{\tau} (\mathbf{store}\ \sigma[\vec{l} \mapsto \vec{v}] \bullet c')}$$

Rule 53 (**Environment**).

$$(a)\quad (\mathbf{env}\ \rho \bullet \mathbf{nil}) \longrightarrow \mathbf{nil} \qquad (b)\ \frac{c \xrightarrow{\ell} c' \quad (\forall \rho \bullet \ell \neq env(\rho))}{(\mathbf{env}\ \rho \bullet c) \xrightarrow{\ell\rho} (\mathbf{env}\ \rho \bullet c')}$$

$$(c)\ \frac{c \xrightarrow{env(\varnothing)} c'}{(\mathbf{store}\ \sigma \bullet c) \xrightarrow{\tau} (\mathbf{store}\ \sigma \bullet c')} \qquad (d)\ \frac{c \xrightarrow{env(\rho_1[\rho_2])} c'}{(\mathbf{env}\ \rho_2 \bullet c) \xrightarrow{env(\rho_1)} (\mathbf{env}\ \rho_2 \bullet c')}$$

Rule 54 (**Procedure declaration**).

$$(\texttt{proc}\ p(\mathbf{ref}\ \vec{z}, \mathbf{val}\ \vec{x}) \mathrel{\widehat{=}} c_1 \bullet c_2) \xrightarrow{env(\rho)} (\mathbf{env}\ \{p \mapsto \mathsf{p}_{\mathsf{cl}}\} \bullet c_2)$$
where
$$\mathsf{p}_{\mathsf{cl}} = (\lambda \vec{L} \bullet \lambda \vec{V} \bullet\ \mathbf{env}\ (\rho \upharpoonright FI(c_1))[(\vec{z} \mapsto \vec{L})] \bullet \texttt{vars}\ (\vec{x} := \vec{V}) \bullet c_1)$$

Fig. A.1. Rules for Plotkin-style locations.

A.2. Variables and the store

The variable declaration command uses the label newlocs$(\vec{l}, \vec{v})$ to add new locations with initial values to the store. The declaration is replaced by a local environment that maps the variable names to the new locations (Rule 51).

The rules for testing and modifying the store are the same as those for modifying a local state in Fig. 4, except that they operate on the label types $l = v$ and $l := v$. However, in addition, the store allows adding new mappings, provided the locations are new (Rule 52). Combining Rules 51 and 52 gives the following derived rule for declaring and initialising a single variable in the context of a global store. The transition hides the newlocs label and, syntax aside, makes the link with Plotkin's rules clearer.

$$\frac{l \notin \text{dom}(\sigma)}{(\mathbf{store}\ \sigma \bullet (\texttt{vars}\ x := v \bullet c)) \longrightarrow (\mathbf{store}\ \sigma[l \mapsto v] \bullet (\mathbf{env}\ \{x \mapsto l\} \bullet c))}$$

A.3. Environments

Rule 53(a) states that an unused environment may be eliminated. Rule 53(b) combines the semantics of renamings and retrieving the value of procedure identifiers in a state (environments are never updated). The label on the subcommand, ℓ, is modified to $\ell\rho$, which is defined by case analysis on ℓ. Assume $\rho(x) = l, y \notin \text{dom}(\rho)$, $\rho(p) = \mathsf{p}_{\mathsf{den}}$, and $q \notin \text{dom}(\rho)$.[5]

$$\begin{aligned} (x = v)\rho &= (l = v) & (y = v)\rho &= (y = v) \\ (x := v)\rho &= (l := v) & (y := v)\rho &= (y := v) \\ (p = \mathsf{p}_{\mathsf{den}})\rho &= \tau & (q = \mathsf{q}_{\mathsf{den}})\rho &= (q = \mathsf{q}_{\mathsf{den}}) \end{aligned}$$

[5] The sets *Loc* and *Val* are distinct, and hence a procedure definition query may always be distinguished from an access of a location.

For any other labels types (with the exception of *env* labels), $\ell\rho = \ell$.

Rule 53(c) states that at the top level of the store, the environment is empty. Rule 53(d) states that the environment for subcommands is accumulated in the normal way, with inner environments overriding the definition of outer environments.

As an example, consider the following inference, which shows the interactions between updates, environments and the store. The inference is justified, top-to-bottom, by Rules 4, 53(b), and the *store* version of Rule 12(a).

$$\frac{\dfrac{x := 2 \xrightarrow{x:=2} \mathbf{nil}}{(\mathbf{env}\ \{x \mapsto l\} \bullet x := 2) \xrightarrow{l:=2} (\mathbf{env}\ \{x \mapsto l\} \bullet \mathbf{nil})}}{(\mathbf{store}\ \{l \mapsto 0\} \bullet (\mathbf{env}\ \{x \mapsto l\} \bullet x := 2)) \xrightarrow{\tau} (\mathbf{store}\ \{l \mapsto 2\} \bullet (\mathbf{env}\ \{x \mapsto l\} \bullet \mathbf{nil}))}$$

A.4. Procedure declarations

Rule 54 gives the elaboration of a procedure declaration p with by-reference parameters $\vec{z}$, by-value parameters $\vec{x}$, and body c_1. Assuming that the label $env(\rho)$ is allowed by the context, which means that the current environment is ρ, the declaration is replaced by an environment that maps p to the closure p_{cl}. Before describing the closure, we first give a procedure denotation using locations and variable declarations.

$$(\lambda \vec{L} \bullet \lambda \vec{V} \bullet \ \mathbf{env}\ (\vec{z} \mapsto \vec{L}) \bullet (\mathtt{vars}\ \vec{x} := \vec{V} \bullet c_1)) \tag{A.1}$$

This is similar to the procedure denotation (15), except that by-reference actual values are locations ($\vec{L}$), and by-value formal parameters are declared using the `vars` command.

The difference between (A.1) and the closure p_{cl} in Rule 54 is that the local environment contains mappings for the free identifiers in c_1, as they appear at the point of declaration. This gives the semantics of static binding. Using Plotkin's notation, $\rho \upharpoonright X$ denotes the subenvironment which is ρ restricted in its domain to X. Furthermore, given a command c_1, the free identifiers in c_1 are given by $FI(c_1)$, where FI may defined straightforwardly following Plotkin.

References

[1] M. Abadi, A. Birrell, T. Harris, M. Isard, Semantics of transactional memory and automatic mutual exclusion, ACM Trans. Program. Lang. Syst. 33 (1) (2011) 2:1–2:50.
[2] L. Aceto, W. Fokkink, C. Verhoef, Structural operational semantics, in: J. Bergstra, A. Ponse, S. Smolka (Eds.), Handbook of Process Algebra, Elsevier Science, 2001, pp. 197–292.
[3] J.W. Backus, F.L. Bauer, J. Green, C. Katz, J. McCarthy, A.J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J.H. Wegstein, A. van Wijngaarden, M. Woodger, P. Naur, Revised report on the algorithm language ALGOL 60, Commun. ACM 6 (1) (1963) 1–17.
[4] J.C.M. Baeten, J.A. Bergstra, Global renaming operators in concrete process algebra, Inform. Comput. 78 (3) (1988) 205–245.
[5] J.C.M. Baeten, J.A. Bergstra, Process algebra with propositional signals, Theor. Comput. Sci. 177 (2) (1997) 381–405.
[6] J.A. Bergstra, J.W. Klop, Process algebra for synchronous communication, Inform. Control 60 (1–3) (1984) 109–137.
[7] R. Colvin, I.J. Hayes, CSP with hierarchical state, in: M. Leuschel, H. Wehrheim (Eds.), Integrated Formal Methods, Lecture Notes in Computer Science, vol. 5423, Springer, 2009, pp. 118–135.
[8] R.J. Colvin, I.J. Hayes, A semantics for Behavior Trees using CSP with specification commands, Sci. Comput. Programming 76 (10) (2011) 891–914.
[9] F. Gadducci, U. Montanari, The tile model, in: G.D. Plotkin, C. Stirling, M. Tofte (Eds.), Proof, Language, and Interaction, Essays in Honour of Robin Milner, The MIT Press, 2000, pp. 133–166.
[10] D. Gries (Ed.), The Science of Programming, Springer-Verlag, 1981
[11] J.F. Groote, M.R. Mousavi, M.A. Reniers, A hierarchy of SOS rule formats, Electron. Notes Theor. Comput. Sci. 156 (1) (2006) 3–25.(Proceedings of the Second Workshop on Structural Operational Semantics (SOS 2005))
[12] C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.
[13] S.S. Ishtiaq, P.W. O'Hearn, BI as an assertion language for mutable data structures, Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), ACM Press, 2001, pp. 14–26.
[14] M. Jaskelioff, N. Ghani, G. Hutton, Modularity and implementation of mathematical operational semantics, Electron. Notes Theor. Comput. Sci. 229 (5) (2011) 75–95.(Proceedings of the Second Workshop on Mathematically Structured Functional Programming (MSFP 2008))
[15] C.B. Jones, Operational semantics: concepts and their expression, Inform. Process. Lett. 88 (1–2) (2003) 27–32.
[16] C.B. Jones, Understanding programming language concepts via operational semantics, in: C. George, Z. Liu, J. Woodcock (Eds.), Domain Modeling and the Duration Calculus, International Training School, Advanced Lecture, Lecture Notes in Computer Science, vol. 4710, Springer, 2007, pp. 177–235.
[17] T. Kleymann, Hoare logic and auxiliary variables, Formal Asp. Comput. 11 (5) (1999) 541–566.
[18] B. Klin, Bialgebras for structural operational semantics: an introduction, 10.1016/j.tcs.2011.03.023Theor. Comput. Sci. (2011)(Corrected Proof)
[19] X. Leroy, H. Grall, Coinductive big-step operational semantics, Inform. and Comput. 207 (2) (2009) 284–304.
[20] J. Matthews, R.B. Findler, Operational semantics for multi-language programs, ACM Trans. Program. Lang. Syst. 31 (3) (2009) 12:1–12:44.
[21] J. McCarthy, Recursive functions of symbolic expressions and their computation by machine, Part I, Commun. ACM 3 (4) (1960) 184–195.
[22] R. Milner, Communication and Concurrency, Prentice Hall, 1989.
[23] R. Milner, M. Tofte, D. MacQueen, The Definition of Standard ML, MIT Press, Cambridge, MA, USA, 1997.
[24] C. Morgan, Programming from Specifications, second ed., Prentice Hall, 1994.
[25] P.D. Mosses, Pragmatics of modular SOS, in: H. Kirchner, C. Ringeissen (Eds.), Algebraic Methodology and Software Technology, 9th International Conference AMAST 2002, Proceedings, Lecture Notes in Computer Science, vol. 2422, Springer, 2002, pp. 21–40.
[26] P.D. Mosses, Modular structural operational semantics, J. Log. Algebr. Program. 60–61 (2004) 195–228.
[27] P.D. Mosses, Exploiting labels in structural operational semantics, Fundam. Inform. 60 (1–4) (2004) 17–31.
[28] P.D. Mosses, M.J. New, Implicit propagation in structural operational semantics, Electron. Notes Theor. Comput. Sci. 229 (4) (2009) 49–66.
[29] M.R. Mousavi, M.A. Reniers, J.F. Groote, Notions of bisimulation and congruence formats for SOS with data, Inform. and Comput. 200 (1) (2005) 107–147.
[30] M.R. Mousavi, M.A. Reniers, J.F. Groote, SOS formats and meta-theory: 20 years after, Theor. Comput. Sci. 373 (3) (2007) 238–272.
[31] S. Owens, A sound semantics for OCaml light, in: S. Drossopoulou (Ed.), European Symposium on Programming (ESOP), Lecture Notes in Computer Science, vol. 4960, Springer, 2008, pp. 1–15.

[32] G.D. Plotkin, A structural approach to operational semantics, Tech. Rep. DAIMI FN-19, Computer Science Department, Aarhus University, 1981.
[33] G.D. Plotkin, A structural approach to operational semantics, J. Log. Algebr. Program. 60–61 (2004) 17–139.
[34] G.D. Plotkin, The origins of structural operational semantics, J. Log. Algebr. Program. 60–61 (2004) 3–15.
[35] J.C. Reynolds, Theories of Programming Languages, Cambridge University Press, 1998.
[36] J.C. Reynolds, Separation logic: a logic for shared mutable data structures, IEEE Symposium on Logic in Computer Science (LICS), IEEE Computer Society, 2002, pp. 55–74.
[37] J.C. Reynolds, An overview of separation logic, in: B. Meyer, J. Woodcock (Eds.), Verified Software: Theories, Tools, Experiments (VSTTE), Lecture Notes in Computer Science, vol. 4171, Springer, 2005, pp. 460–469.
[38] A. Roscoe, The Theory and Practice of Concurrency, Prentice Hall, 1998.
[39] S. Schneider, Concurrent and Real-time Systems: The CSP Approach, Wiley, 2000.
[40] R.W. Sebesta, Concepts of Programming Languages, ninth ed., Addison-Wesley, 2008.
[41] D. Turi, G.D. Plotkin, Towards a mathematical operational semantics, IEEE Symposium on Logic in Computer Science (LICS), IEEE Computer Society, 1997, pp. 280–291.
[42] D.A. van Beek, K.L. Man, M.A. Reniers, J.E. Rooda, R.R.H. Schiffelers, Syntax and consistent equation semantics of hybrid χ, J. Log. Algebr. Program. 68 (1–2) (2006) 129–210.
[43] D. von Oheimb, Hoare logic for mutual recursion and local variables, in: C.P. Rangan, V. Raman, R. Ramanujam (Eds.), Foundations of Software Technology and Theoretical Computer Science (FSTTCS), Lecture Notes in Computer, vol. 1738, Springer, 1999, pp. 168–180.
[44] G. Winskel, The Formal Semantics of Programming Languages: An Introduction, MIT Press, 1993.